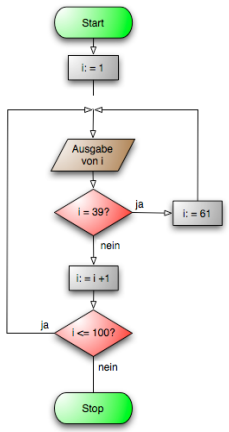
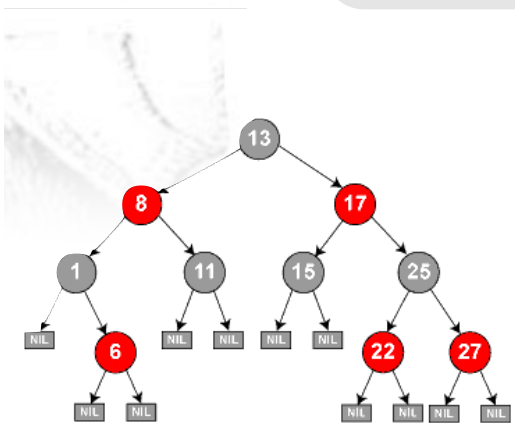
Algorithmen und Datenstrukturen





Version 1.28 vom 03. Dezember 2023

Fachhochschule Aachen

Lehrgebiet für Angewandte Informatik

Prof. Dr. rer. nat. Jörg Striegnitz

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Bedeutung von Sortierverfahren

- **Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet** [Mehlhorn, 1988, S. 36]

© 2006-2023 Jörg Striegnitz

Bedeutung von Sortierverfahren

- **Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet** [Mehlhorn, 1988, S. 36]

- **Die wichtigsten Anwendungen vom Sortieren sind;**
  - **Zusammenbringen von zusammengehörigen Dingen**; z.B:.
    - **1. Liste**: Zuordnung von Personen zu Firmen
      eine Person darf für mehrere Firmen arbeiten
    - **2. Liste**: Pro Firma: Zuordnung eines Gehalts zu einer Person
    - **Vereinigung der Listen und Sortieren nach Namen**: Gesamteinkünfte einer Person liegen gruppiert zusammen
  - **Finden von Duplikaten**
  - **Sortieren hilft beim Suchen**
    siehe Folgekapitel

© 2006-2023 Jörg Striegnitz

Bedeutung von Sortierverfahren

- **Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet** [Mehlhorn, 1988, S. 36]

- **Die wichtigsten Anwendungen vom Sortieren sind;**
  - **Zusammenbringen von zusammengehörigen Dingen**; z.B:.
    - **1. Liste**: Zuordnung von Personen zu Firmen
      eine Person darf für mehrere Firmen arbeiten
    - **2. Liste**: Pro Firma: Zuordnung eines Gehalts zu einer Person
    - **Vereinigung der Listen und Sortieren nach Namen**: Gesamteinkünfte einer Person liegen gruppiert zusammen
  - **Finden von Duplikaten**
  - **Sortieren hilft beim Suchen**
    siehe Folgekapitel

- **Viele Aufgaben sind mit Sortierproblem verwandt; z.B.:**
  - Bestimmung des Median - das Element an der mittleren Position einer sortierten Folge
  - Bestimmung der k kleinsten/größten Elemente

© 2006-2023 Jörg Striegnitz

Bedeutung von Sortierverfahren

- **Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet** [Mehlhorn, 1988, S. 36]

- **Die wichtigsten Anwendungen vom Sortieren sind;**
  - **Zusammenbringen von zusammengehörigen Dingen**; z.B:.
    - **1. Liste**: Zuordnung von Personen zu Firmen
      eine Person darf für mehrere Firmen arbeiten
    - **2. Liste**: Pro Firma: Zuordnung eines Gehalts zu einer Person
    - **Vereinigung der Listen und Sortieren nach Namen**: Gesamteinkünfte einer Person liegen gruppiert zusammen
  - **Finden von Duplikaten**
  - **Sortieren hilft beim Suchen**
    siehe Folgekapitel

- **Viele Aufgaben sind mit Sortierproblem verwandt; z.B.:**
  - Bestimmung des Median - das Element an der mittleren Position einer sortierten Folge
  - Bestimmung der k kleinsten/größten Elemente

- **Daher: Hohe Effizienzansprüche an Algorithmen!**

© 2006-2023 Jörg Striegnitz

Übersicht:

- **Hier behandelt:**
  - **Elementare Sortierverfahren**
    - haben im Mittel (in der Größe der Eingabe) ***quadratische Laufzeit***
      - Selection Sort (Sortieren durch Auswahl - *Wiederholung*)
      - Insertion Sort (Sortieren durch Einfügen)
      - Bubble Sort
      - Bucket Sort
  - **Höhere Sortierverfahren**
    - haben im Mittel ***überlineare Laufzeit*** $O(n \log n)$
      - QuickSort
      - HeapSort
      - MergeSort

© 2006-2023 Jörg Striegnitz

Voraussetzungen

- **Für nachfolgende Betrachtungen sei stets eine Folge**

  `a[1] ... a[n]`

  von Datensätzen gegeben.

© 2006-2023 Jörg Striegnitz

Voraussetzungen

- **Für nachfolgende Betrachtungen sei stets eine Folge**

  `a[1] ... a[n]`

  von Datensätzen gegeben.

- **Jeder Datensatz besitzt eine Schlüsselkomponente**

  `a[i].key` (i=1,..,n)

© 2006-2023 Jörg Striegnitz

Voraussetzungen

- **Für nachfolgende Betrachtungen sei stets eine Folge**
  `a[1] ... a[n]`
  von Datensätzen gegeben.

- **Jeder Datensatz besitzt eine Schlüsselkomponente**
  `a[i].key` (i=1,..,n)

- **Datensätze dürfen weitere Attribute enthalten**
  Aber die Sortierung erfolgt ausschließlich nach der Schlüsselkomponente

© 2006-2023 Jörg Striegnitz

Voraussetzungen

- **Für nachfolgende Betrachtungen sei stets eine Folge**
  `a[1] ... a[n]`
  von Datensätzen gegeben.

- **Jeder Datensatz besitzt eine Schlüsselkomponente**
  `a[i].key` (i=1,..,n)

- **Datensätze dürfen weitere Attribute enthalten**
  Aber die Sortierung erfolgt ausschließlich nach der Schlüsselkomponente

- **Für die Sortierung muss auf der Menge aller Schlüssel eine Ordnung definiert sein**

© 2006-2023 Jörg Striegnitz

Ordnungen ...

© 2006-2023 Jörg Striegnitz

Ordnungen ...

D Halbordnung / totale Ordnung

Es sei M eine nichtleere Menge und "$\leq$" $\subseteq M \times M$ eine binäre Relation über M. Die Relation $\leq$ heißt eine **Halbordnung** über M, genau dann wenn, $\leq$ die folgenden Eigenschaften erfüllt:

Reflexivität: $\forall x \in M : x \leq x$

Transitivität: $\forall x, y, z \in M : x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$

Antisymmetrie: $\forall x, y \in M : x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$

© 2006-2023 Jörg Striegnitz

D Halbordnung / totale Ordnung

Es sei M eine nichtleere Menge und "$\leq$" $\subseteq M \times M$ eine binäre Relation über M. Die Relation $\leq$ heißt eine **Halbordnung** über M, genau dann wenn, $\leq$ die folgenden Eigenschaften erfüllt:

Reflexivität: $\forall x \in M : x \leq x$
Transitivität: $\forall x, y, z \in M : x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
Antisymmetrie: $\forall x, y \in M : x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$

Sei $\leq$ eine Halbordnung über M. Der **strikte Anteil** "$<$" von $\leq$ ist definiert als:

$$x < y \Leftrightarrow x \leq y \ \wedge \ x \neq y$$

© 2006-2023 Jörg Striegnitz

Ordnungen ...

D Halbordnung / totale Ordnung

Es sei M eine nichtleere Menge und "$\leq$" $\subseteq M \times M$ eine binäre Relation über M. Die Relation $\leq$ heißt eine **Halbordnung** über M, genau dann wenn, $\leq$ die folgenden Eigenschaften erfüllt:

Reflexivität: $\forall x \in M : x \leq x$
Transitivität: $\forall x, y, z \in M : x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
Antisymmetrie: $\forall x, y \in M : x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$

Sei $\leq$ eine Halbordnung über M. Der **strikte Anteil** "$<$" von $\leq$ ist definiert als:

$$x < y \Leftrightarrow x \leq y \ \wedge \ x \neq y$$

Gilt für $\leq$ ferner

$$\forall x, y \in M : x < y \ \vee \ x = y \ \vee \ y < x$$

so heisst $\leq$ **totale Ordnung** über M.

© 2006-2023 Jörg Striegnitz

Ordnungen ...

D Halbordnung / totale Ordnung

Es sei M eine nichtleere Menge und "$\leq$" $\subseteq M \times M$ eine binäre Relation über M. Die Relation $\leq$ heißt eine **Halbordnung** über M, genau dann wenn, $\leq$ die folgenden Eigenschaften erfüllt:

Reflexivität: $\forall x \in M : x \leq x$
Transitivität: $\forall x, y, z \in M : x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
Antisymmetrie: $\forall x, y \in M : x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$

Sei $\leq$ eine Halbordnung über M. Der **strikte Anteil** "$<$" von $\leq$ ist definiert als:

$$x < y \Leftrightarrow x \leq y \ \wedge \ x \neq y$$

Gilt für $\leq$ ferner

$$\forall x, y \in M : x < y \ \vee \ x = y \ \vee \ y < x$$

so heisst $\leq$ **totale Ordnung** über M.

Halbordnungen nennt man oft auch **partielle Ordnung**.; totale Ordnungen sind auch unter dem Namen **lineare Ordnung** bekannt.

© 2006-2023 Jörg Striegnitz

Das Sortierproblem formal:

D **Sortierproblem**

Gegeben seien eine Folge $\mathtt{a}[1], \ldots, \mathtt{a}[\mathtt{n}]$ von Datensätzen mit einer Schlüsselkomponenten $\mathtt{a}[\mathtt{i}].\mathtt{key}$ und eine totale Ordnung $\leq$ über der Menge aller Schlüssel. Das **Sortierproblem** besteht darin, eine Permutation $\pi = (\pi_1, \cdots, \pi_n)$ des Indexbereichs $1, \cdots, n$ zu finden, so dass gilt:

$$\mathtt{a}[\pi_1].\mathtt{key} \leq \mathtt{a}[\pi_2].\mathtt{key} \leq \cdots \leq \mathtt{a}[\pi_n].\mathtt{key}$$

© 2006-2023 Jörg Striegnitz

Das Sortierproblem formal:

D **Sortierproblem**

Gegeben seien eine Folge $\mathtt{a}[1], \ldots, \mathtt{a}[\mathtt{n}]$ von Datensätzen mit einer Schlüsselkomponenten $\mathtt{a}[\mathtt{i}].\mathtt{key}$ und eine totale Ordnung $\leq$ über der Menge aller Schlüssel. Das **Sortierproblem** besteht darin, eine Permutation $\pi = (\pi_1, \cdots, \pi_n)$ des Indexbereichs $1, \cdots, n$ zu finden, so dass gilt:

$$\mathtt{a}[\pi_1].\mathtt{key} \leq \mathtt{a}[\pi_2].\mathtt{key} \leq \cdots \leq \mathtt{a}[\pi_n].\mathtt{key}$$

Beachte: da wir eine Folge und keine Menge voraussetzen, dürfen Schlüssel (und ganze Datensätze) **mehrfach** auftreten!

© 2006-2023 Jörg Striegnitz

Das Sortierproblem formal:

D **Sortierproblem**

Gegeben seien eine Folge $\mathtt{a}[1], \ldots, \mathtt{a}[\mathtt{n}]$ von Datensätzen mit einer Schlüsselkomponenten $\mathtt{a}[\mathtt{i}].\mathtt{key}$ und eine totale Ordnung $\leq$ über der Menge aller Schlüssel. Das **Sortierproblem** besteht darin, eine Permutation $\pi = (\pi_1, \cdots, \pi_n)$ des Indexbereichs $1, \cdots, n$ zu finden, so dass gilt:

$$\mathtt{a}[\pi_1].\mathtt{key} \leq \mathtt{a}[\pi_2].\mathtt{key} \leq \cdots \leq \mathtt{a}[\pi_n].\mathtt{key}$$

Beachte: da wir eine Folge und keine Menge voraussetzen, dürfen Schlüssel (und ganze Datensätze) **mehrfach** auftreten!

Liste	**Schlüsselelement**	**Ordnung**
Telefonbuch	Nachname	lexikographische Ordnung
Klausurergebnisse	Punktezahl	$\leq$ auf $\mathbb{R}$
Lexikon	Stichwort	lexikographische Ordnung
Studentenverzeichnis	Matrikelnummer	$\leq$ auf $\mathbb{N}$
Entfernungstabelle	Distanz	$\leq$ auf $\mathbb{R}$
Fahrplan	Abfahrtszeit	„früher als“

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**

  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**
  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**
  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren
  - **intern** (alle Daten im Speicher) / **extern** (z.B. Festplatte)

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**
  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren
  - **intern** (alle Daten im Speicher) / **extern** (z.B. Festplatte)
  - **direkt / indirekt** (d.h. mit Zeigern oder Array-Indizes)

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**
  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren
  - **intern** (alle Daten im Speicher) / **extern** (z.B. Festplatte)
  - **direkt / indirekt** (d.h. mit Zeigern oder Array-Indizes)
  - ***in situ*** (ohne Hilfsfelder) oder nicht

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**

  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren
  - **intern** (alle Daten im Speicher) / **extern** (z.B. Festplatte)
  - **direkt / indirekt** (d.h. mit Zeigern oder Array-Indizes)
  - ***in situ*** (ohne Hilfsfelder) oder nicht
  - **allgemein / speziell**
    z.B. stellt `Bucket Sort` Anforderungen an die Schlüsselmenge

© 2006-2023 Jörg Striegnitz

Merkmale von Sortierverfahren

- **Sortierverfahren können nach verschiedenen Kriterien klassifiziert werden:**

  - **Sortiermethode:** algorithmische Idee
    z.B. Sortieren durch Selektion, durch Verteilen, durch Mischen
  - **Effizienz**: $O(n^2)$ für elementare und $O(n \log n)$ für höhere Sortierverfahren
  - **intern** (alle Daten im Speicher) / **extern** (z.B. Festplatte)
  - **direkt / indirekt** (d.h. mit Zeigern oder Array-Indizes)
  - ***in situ*** (ohne Hilfsfelder) oder nicht
  - **allgemein / speziell**
    z.B. stellt `Bucket Sort` Anforderungen an die Schlüsselmenge
  - **stabil** (Reihenfolge von Elementen mit gleichem Schlüssel bleibt erhalten) oder nicht

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- **4.1.1. Selection Sort**
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Sortieren durch Auswählen (`SelectionSort`)

```
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```


© 2006-2023 Jörg Striegnitz

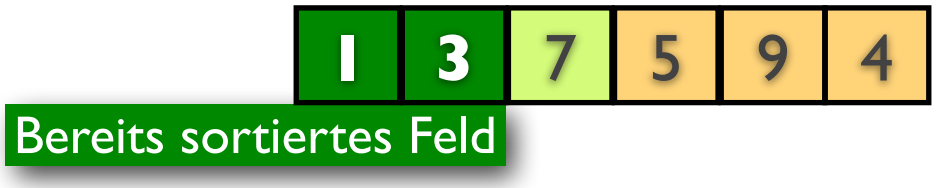
Sortieren durch Auswählen (`SelectionSort`)

```
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```

© 2006-2023 Jörg Striegnitz

Sortieren durch Auswählen (`SelectionSort`)

```
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```



© 2006-2023 Jörg Striegnitz

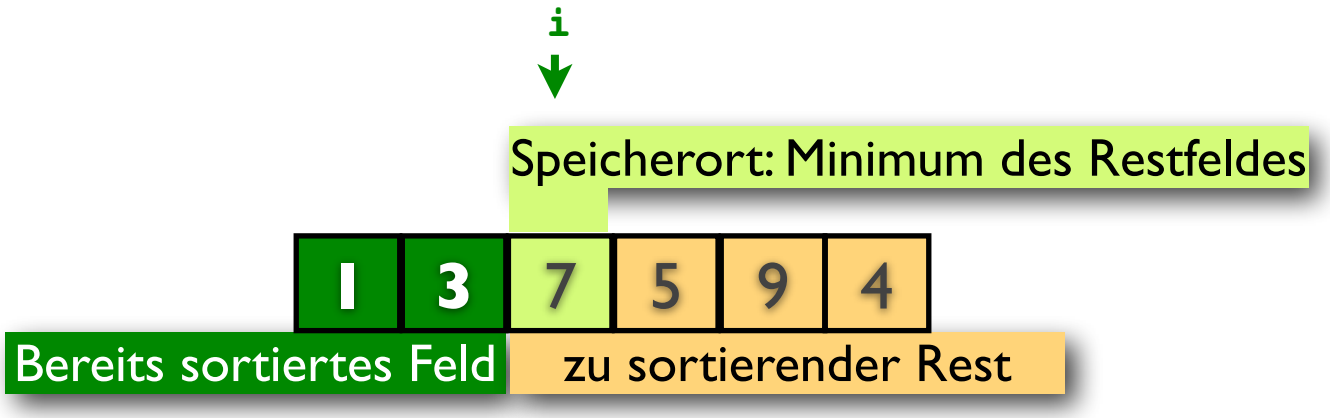
Sortieren durch Auswählen (`SelectionSort`)

```
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```

© 2006-2023 Jörg Striegnitz

Sortieren durch Auswählen (`SelectionSort`)

```
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```



© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Achtung: Für jeden Tausch müssen ***drei Datensätze bewegt*** werden!

© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Achtung: Für jeden Tausch müssen ***drei Datensätze bewegt*** werden!

Weitere Eigenschaften von `Selection Sort`:

© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Achtung: Für jeden Tausch müssen ***drei Datensätze bewegt*** werden!

Weitere Eigenschaften von `Selection Sort`:

- *in situ* Verfahren

© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Achtung: Für jeden Tausch müssen ***drei Datensätze bewegt*** werden!

Weitere Eigenschaften von `Selection Sort`:

- *in situ* Verfahren
- Komplexität in *best*, *worst* und *average case* identisch
  daher: unabhängig von Vorsortierung

© 2006-2023 Jörg Striegnitz

Komplexität `SelectionSort`

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Achtung: Für jeden Tausch müssen ***drei Datensätze bewegt*** werden!

Weitere Eigenschaften von `Selection Sort`:

- *in situ* Verfahren
- Komplexität in *best*, *worst* und *average case* identisch
  daher: unabhängig von Vorsortierung
- `Selection Sort` ist in seiner Grundform **nicht stabil**
  (lässt sich aber mit zusätzlichem Aufwand stabil implementieren)

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- **4.1.2. Insertion Sort**
- 4.1.3. Bubble Sort

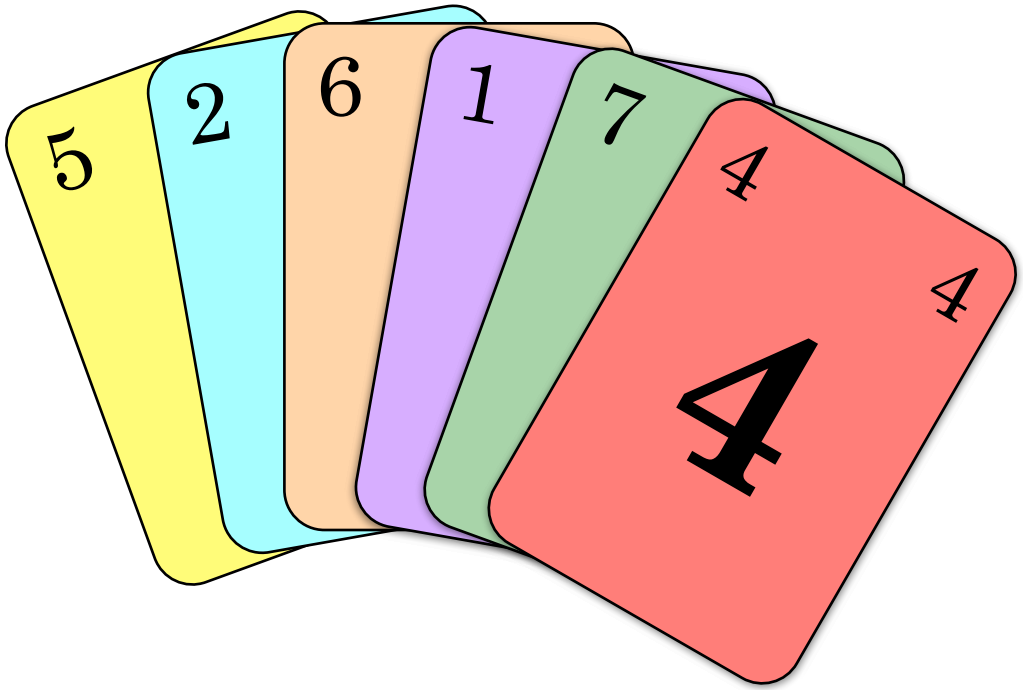
4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3.Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

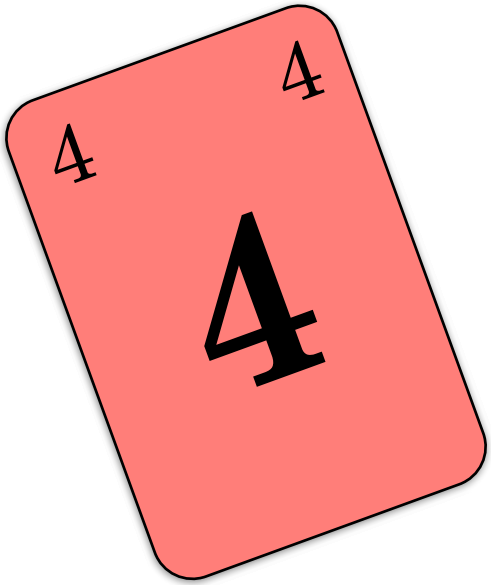
- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein

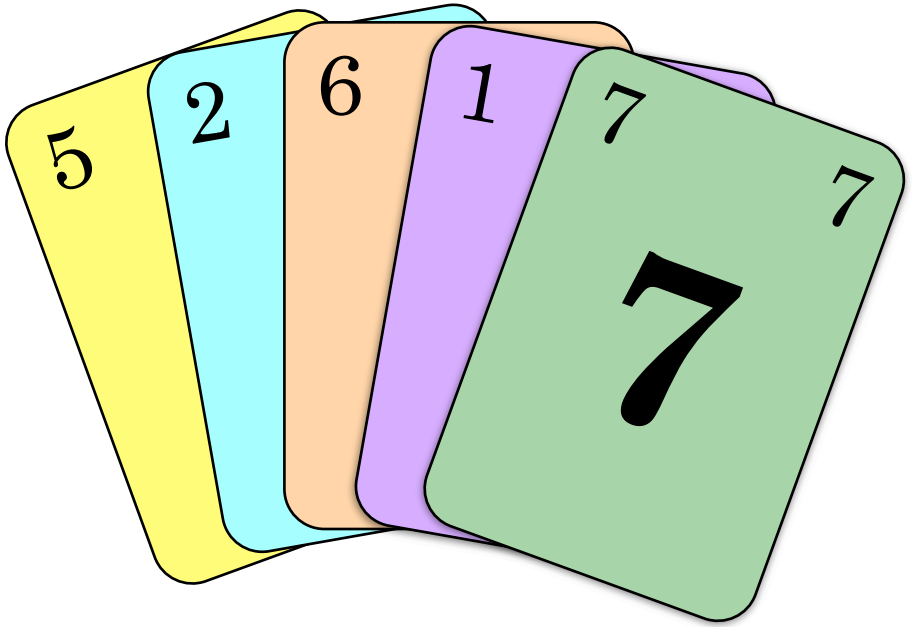


© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein

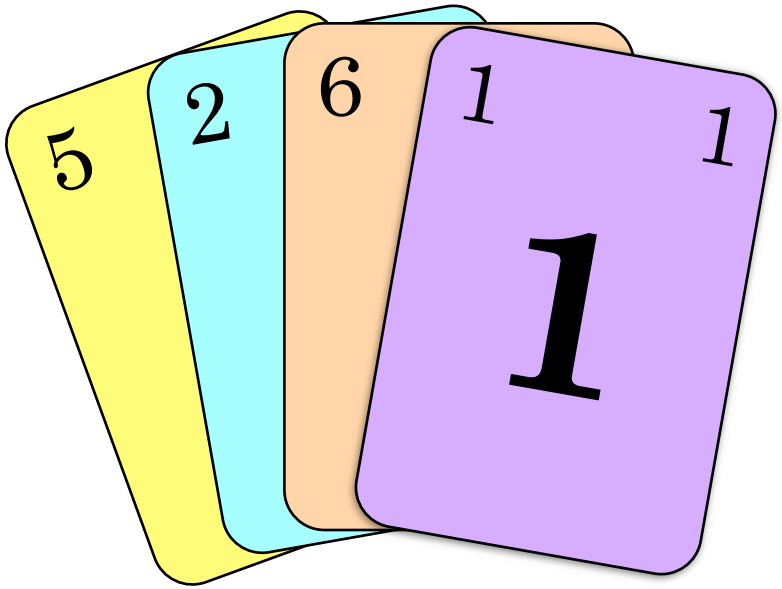




© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

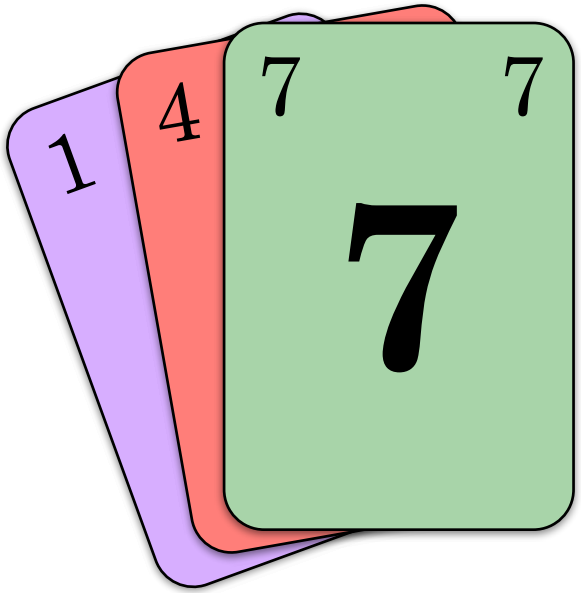
- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein

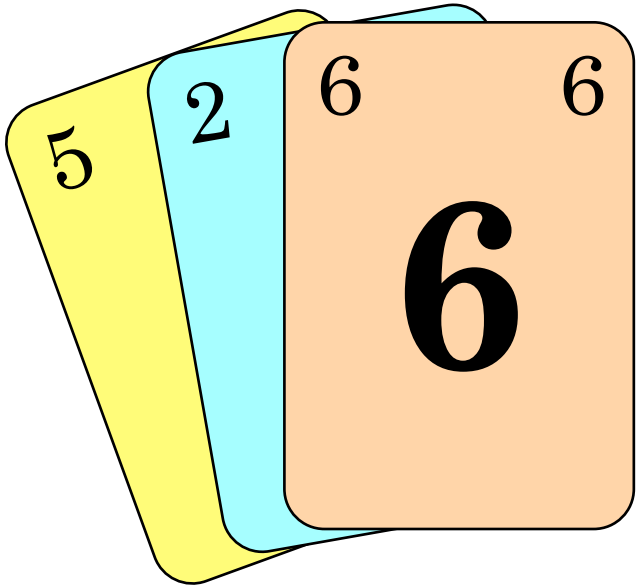


© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein

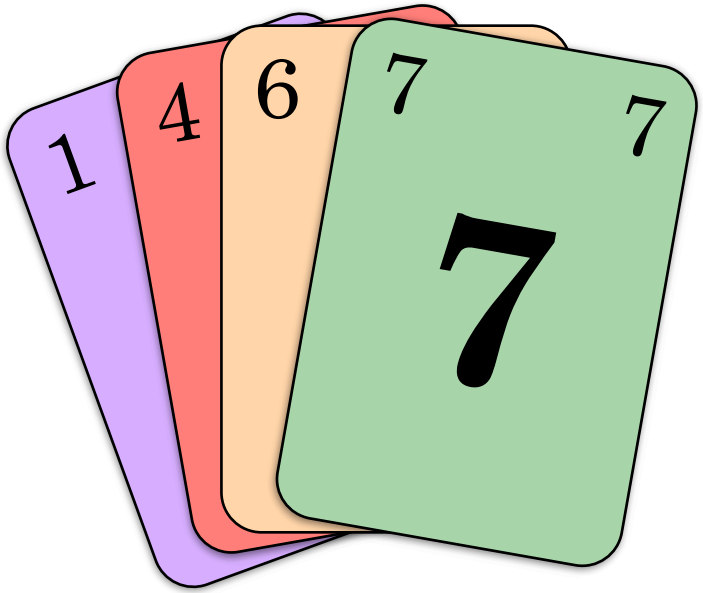


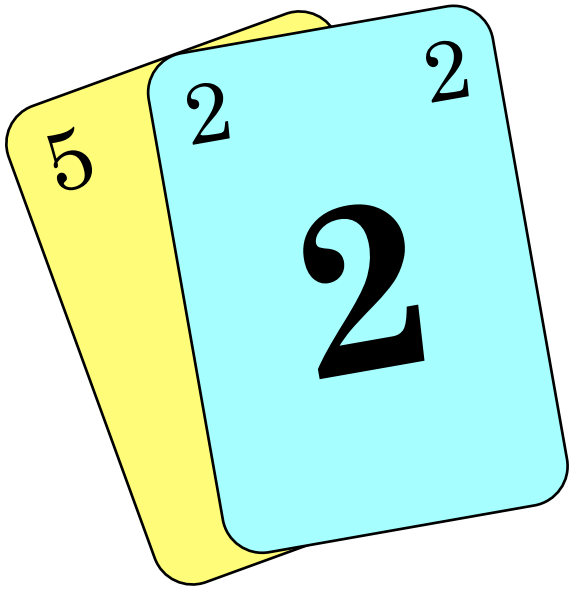


© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein

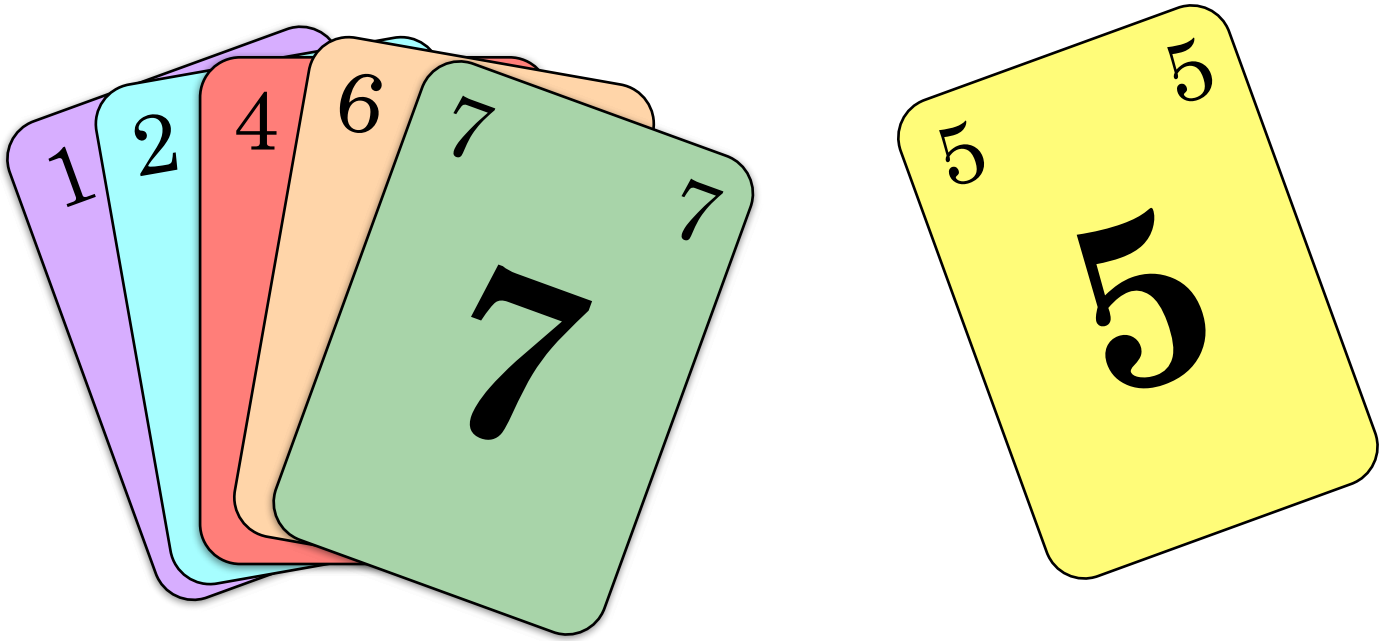




© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

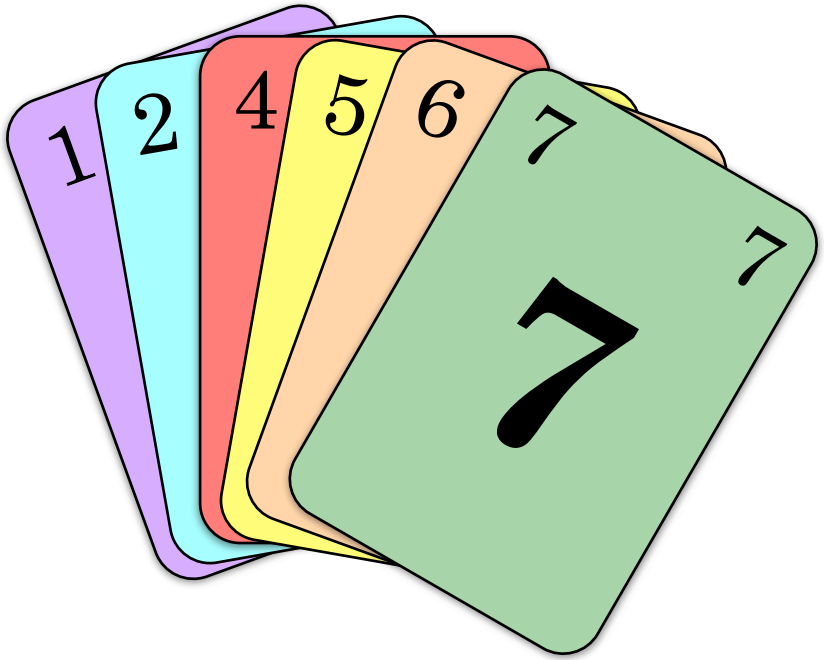
- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein



© 2006-2023 Jörg Striegnitz

Sortieren durch Einfügen (`Insertion Sort`)

- **Vorbild: manuelles Sortieren**
- **Grundidee (am Beispiel Sortieren von Spielkarten)**
  - Starte mit der ersten Karte einen neuen Stapel
  - Nimm jeweils die nächste Karte des Originalstapels und füge sie an der richtigen Stelle in den neuen Stapel ein



© 2006-2023 Jörg Striegnitz

Insertion Sort: Implementierung (JAVA)

```
public void insertionSort(int feld[]) {
   for(int i=1; i <feld.length; i++) {
     int m = feld[i]; // einzufügendes Element
     int j; // brauchen wir nach Schleifenaustritt!
     // Für alle Elemente links von i-ten Element ...
     for (j=i; j>0; j--) { // Suche von rechts nach links!
       if (m >= feld[j-1]) break;
       // größere Elemente nach hinten schieben
       feld[j] = feld[j-1];
     }
     // m an freien Platz einfügen
     feld[j] = m;
   }
 }
```


© 2006-2023 Jörg Striegnitz

Insertion Sort: Implementierung (JAVA)

```
public void insertionSort(int feld[]) {
   for(int i=1; i <feld.length; i++) {
     int m = feld[i]; // einzufügendes Element
     int j; // brauchen wir nach Schleifenaustritt!
     // Für alle Elemente links von i-ten Element ...
     for (j=i; j>0; j--) { // Suche von rechts nach links!
       if (m >= feld[j-1]) break;
       // größere Elemente nach hinten schieben
       feld[j] = feld[j-1];
     }
     // m an freien Platz einfügen
     feld[j] = m;
   }
 }
```

`Insertion Sort` ist ein **stabiles *in situ*** Verfahren

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität I

- **Vergleiche**

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität I

- **Vergleiche**
  - Das **Einfügen** des Elementes $a[i]$ in die bereits sortierte Anfangsfolge $a[0]$ ... $a[i\text{-}1]$ erfordert mindestens einen Vergleich, höchstens jedoch i Vergleiche; im Mittel also $i/2$ Vergleiche

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität I

- **Vergleiche**
  - Das **Einfügen** des Elementes $a[i]$ in die bereits sortierte Anfangsfolge $a[0]$ ... $a[i\text{-}1]$ erfordert mindestens einen Vergleich, höchstens jedoch i Vergleiche; im Mittel also $i/2$ Vergleiche
  - ***Best Case***: Bei vollständig vorsortierten Folgen ergeben sich $N\text{-}1$ Vergleiche

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität I

- **Vergleiche**

  - Das **Einfügen** des Elementes $a[i]$ in die bereits sortierte Anfangsfolge $a[0]$ ... $a[i\text{-}1]$ erfordert mindestens einen Vergleich, höchstens jedoch i Vergleiche; im Mittel also $i/2$ Vergleiche
  - ***Best Case***: Bei vollständig vorsortierten Folgen ergeben sich $N\text{-}1$ Vergleiche
  - ***Worst Case***: Bei umgekehrt sortierten Folgen gilt für die Anzahl der Vergleiche:

$$\begin{aligned}\sum_{i=1}^{N-1} i &= \frac{N\,(N-1)}{2} \\ &= \frac{N^2}{2} - \frac{N}{2} \approx \frac{N^2}{2}\end{aligned}$$

© 2006-2023 Jörg Striegnitz

Insertion Sort: Komplexität I

- **Vergleiche**

  - Das **Einfügen** des Elementes $a[i]$ in die bereits sortierte Anfangsfolge $a[0]$ ... $a[i\text{-}1]$ erfordert mindestens einen Vergleich, höchstens jedoch i Vergleiche; im Mittel also $i/2$ Vergleiche
  - ***Best Case***: Bei vollständig vorsortierten Folgen ergeben sich $N\text{-}1$ Vergleiche
  - ***Worst Case***: Bei umgekehrt sortierten Folgen gilt für die Anzahl der Vergleiche:

$$\begin{aligned}\sum_{i=1}^{N-1} i &= \frac{N\,(N-1)}{2} \\ &= \frac{N^2}{2} - \frac{N}{2} \approx \frac{N^2}{2}\end{aligned}$$

  - ***Average Case***: Jedes Element wandert in etwa in die Mitte des sortierten Feldes:

$$\frac{1}{2} \cdot \sum_{i=1}^{N-1} i = \frac{N^2}{4} - \frac{N}{4} \approx \frac{N^2}{4}$$

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität II

- **Bewegungen**

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität II

- **Bewegungen**
  - Betrachte Schleifendurchgang i:

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität II

- **Bewegungen**
  - Betrachte Schleifendurchgang i:
    - Ein Kopiervorgang in Hilfsvariable m
    - Anschließend Vergleich mit höchstens i, mindestens aber einem Schlüssel - im Mittel $i/2$ Vergleiche. Bis auf das letzte Vergleichselement werden alle nach rechts geschoben (jeweils eine Bewegung)
    - Eine Bewegung von Hilfsvariable an Zielposition
    - Also: Im Schleifendurchgang i mindestens zwei, höchstens $i+1$ Bewegungen und im Mittel $(i+1)/2$

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität II

- **Bewegungen**
  - Betrachte Schleifendurchgang i:
    - Ein Kopiervorgang in Hilfsvariable m
    - Anschließend Vergleich mit höchstens i, mindestens aber einem Schlüssel - im Mittel $i/2$ Vergleiche. Bis auf das letzte Vergleichselement werden alle nach rechts geschoben (jeweils eine Bewegung)
    - Eine Bewegung von Hilfsvariable an Zielposition
    - Also: Im Schleifendurchgang i mindestens zwei, höchstens $i+1$ Bewegungen und im Mittel $(i+1)/2$
  - ***Best Case***: Bei vollständig vorsortierten Folgen $2(N-1)$ Bewegungen

© 2006-2023 Jörg Striegnitz

`Insertion Sort`: Komplexität II

- **Bewegungen**
  - Betrachte Schleifendurchgang i:
    - Ein Kopiervorgang in Hilfsvariable m
    - Anschließend Vergleich mit höchstens i, mindestens aber einem Schlüssel - im Mittel $i/2$ Vergleiche. Bis auf das letzte Vergleichselement werden alle nach rechts geschoben (jeweils eine Bewegung)
    - Eine Bewegung von Hilfsvariable an Zielposition
    - Also: Im Schleifendurchgang i mindestens zwei, höchstens $i+1$ Bewegungen und im Mittel $(i+1)/2$
  - ***Best Case***: Bei vollständig vorsortierten Folgen $2(N\text{-}1)$ Bewegungen
  - ***Worst Case***: Bei umgekehrt sortierten Folgen $N^2/2$ Bewegungen

© 2006-2023 Jörg Striegnitz

Insertion Sort: Komplexität II

- **Bewegungen**
  - Betrachte Schleifendurchgang i:
    - Ein Kopiervorgang in Hilfsvariable m
    - Anschließend Vergleich mit höchstens i, mindestens aber einem Schlüssel - im Mittel $i/2$ Vergleiche. Bis auf das letzte Vergleichselement werden alle nach rechts geschoben (jeweils eine Bewegung)
    - Eine Bewegung von Hilfsvariable an Zielposition
    - Also: Im Schleifendurchgang i mindestens zwei, höchstens $i+1$ Bewegungen und im Mittel $(i+1)/2$
  - ***Best Case***: Bei vollständig vorsortierten Folgen $2(N-1)$ Bewegungen
  - ***Worst Case***: Bei umgekehrt sortierten Folgen $N^2/2$ Bewegungen
  - ***Average Case***: Etwa $N^2/4$ Bewegungen

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- **4.1.3. Bubble Sort**

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3.Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Bubble Sort

- **Auch bekannt als „Sortieren durch lokales Vertauschen“**
- **Idee: vertausche Nachbarn, die falsche Reihenfolge haben und zwar so lange, bis das Feld vollständig sortiert ist.**
- **Methode:**
  - Durchlaufe Feld von 1 bis N
  - Falls $\mathrm{a[i\text{-}1]>a[i]}$ vertausche die Elemente
  - Konsequenz: größtes Element wandert nach ganz rechts an Position N („steigt auf wie eine Blase“)
  - Wiederhole das Verfahren für den Bereich $1 \ldots N\text{-}1$

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1

3	10	7	5	9	4	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1

3	10	7	5	9	4	1

3	7	10	5	9	4	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1

3	10	7	5	9	4	1

3	7	10	5	9	4	1

3	7	5	10	9	4	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1
3	10	7	5	9	4	1
3	7	10	5	9	4	1
3	7	5	10	9	4	1
3	7	5	9	10	4	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1
3	10	7	5	9	4	1
3	7	10	5	9	4	1
3	7	5	10	9	4	1
3	7	5	9	10	4	1
3	7	5	9	4	10	1

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 1. Iteration

10	3	7	5	9	4	1
3	10	7	5	9	4	1
3	7	10	5	9	4	1
3	7	5	10	9	4	1
3	7	5	9	10	4	1
3	7	5	9	4	10	1
3	7	5	9	4	1	**10**

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10

3	7	5	9	4	1	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10

3	7	5	9	4	1	10

3	5	7	9	4	1	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10
3	7	5	9	4	1	10
3	5	7	9	4	1	10
3	5	7	9	4	1	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10
3	7	5	9	4	1	10
3	5	7	9	4	1	10
3	5	7	9	4	1	10
3	5	7	4	9	1	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10

3	7	5	9	4	1	10

3	5	7	9	4	1	10

3	5	7	9	4	1	10

3	5	7	4	9	1	10

3	5	7	4	1	9	10

© 2006-2023 Jörg Striegnitz

Bubble Sort: Beispiel

B 2. Iteration

3	7	5	9	4	1	10
3	7	5	9	4	1	10
3	5	7	9	4	1	10
3	5	7	9	4	1	10
3	5	7	4	9	1	10
3	5	7	4	1	9	10

Das jeweils größte Element wandert wie eine Blase (*Bubble*) nach oben (rechts)

© 2006-2023 Jörg Striegnitz

Bubble Sort: Implementierung (JAVA)

```
public void bubbleSort(int feld[]) {
    for (int i=feld.length-1 ; i>=0 ; --i)
      for (int j=1; j<=i ; ++j)
        if (feld[j]<feld[j-1]) {
          int t = feld[j];
          feld[j]=feld[j-1];
          feld[j-1]=t;
        }
}
```


© 2006-2023 Jörg Striegnitz

Bubble Sort: Implementierung (JAVA)

```
public void bubbleSort(int feld[]) {
    for (int i=feld.length-1 ; i>=0 ; --i)
      for (int j=1; j<=i ; ++j)
        if (feld[j]<feld[j-1]) {
          int t = feld[j];
          feld[j]=feld[j-1];
          feld[j-1]=t;
        }
}
```

Bubble Sort ist ein **stabiles *in situ*** Verfahren

© 2006-2023 Jörg Striegnitz

`Bubble Sort`: Komplexitätsanalyse I

- **Vergleiche**
  - Die Anzahl der Vergleiche ist unabhängig von der Vorsortierung!
  - *worst*, *best* und *average case* sind daher identisch
  - Im i-ten Schleifendurchlauf (i=N-1, N-2, ... , 1) enthält die noch unsortierte Folge N-i+1 Elemente, für die N-i Vergleiche nötig sind.
  - Zum Sortieren der Gesamtfolge sind N-1 Schritte erforderlich
  - Die Anzahl der Vergleiche wächst damit quadratisch mit der Anzahl der Elemente:

© 2006-2023 Jörg Striegnitz

Bubble Sort: Komplexitätsanalyse I

- **Vergleiche**
  - Die Anzahl der Vergleiche ist unabhängig von der Vorsortierung!
  - *worst*, *best* und *average case* sind daher identisch
  - Im i-ten Schleifendurchlauf (i=N-1, N-2, ... , 1) enthält die noch unsortierte Folge N-i+1 Elemente, für die N-i Vergleiche nötig sind.
  - Zum Sortieren der Gesamtfolge sind N-1 Schritte erforderlich
  - Die Anzahl der Vergleiche wächst damit quadratisch mit der Anzahl der Elemente:

$$\begin{aligned}\sum_{i=1}^{N-1}(N-i) &= \sum_{i=1}^{N-1} i \\ &= \frac{N\cdot(N-1)}{2}\end{aligned}$$

© 2006-2023 Jörg Striegnitz

`Bubble Sort`: Komplexitätsanalyse II

- **Bewegungen**
  - **Best Case:** (Feld ist sortiert) keine Bewegungen
  - **Worst Case**: etwa $3N^2/2$ Bewegungen
  - **Average Case**: etwa $3N^2/4$ Bewegungen

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

- **Bewegungen**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

- **Bewegungen**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

Bewertung:

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

- **Bewegungen**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

Bewertung:

`Bubble Sort:` Ineffizient (immer $N^2/2$ Vergleiche)

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

- **Bewegungen**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

Bewertung:

`Bubble Sort:` Ineffizient (immer $N^2/2$ Vergleiche)

`Insertion Sort:` gut für fast sortierte Folgen

© 2006-2023 Jörg Striegnitz

Elementare Sortierverfahren - Gegenüberstellung

- **Vergleiche**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

- **Bewegungen**

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

Bewertung:

`Bubble Sort:` Ineffizient (immer $N^2/2$ Vergleiche)

`Insertion Sort:` gut für fast sortierte Folgen

`Selection Sort:` gut für große Datensätze, da Zahl der Vertauschungen konstant

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**

- **Dann: Sortiere Verweise anstelle von Datensätzen**

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**
- **Dann: Sortiere Verweise anstelle von Datensätzen**
- **Verfahren:**

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**
- **Dann: Sortiere Verweise anstelle von Datensätzen**
- **Verfahren:**
  - Verwende ein Index-Feld `p[0..N-1]`, das mit `p[i]=i` initialisiert wird (`i=1,...,N`)

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**
- **Dann: Sortiere Verweise anstelle von Datensätzen**
- **Verfahren:**
  - Verwende ein Index-Feld `p[0..N-1]`, das mit `p[i]=i` initialisiert wird (`i=1,...,N`)
  - Für Vergleiche erfolgt der Zugriff auf einen Record über `a[p[i]]`

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**
- **Dann: Sortiere Verweise anstelle von Datensätzen**
- **Verfahren:**
  - Verwende ein Index-Feld `p[0..N-1]`, das mit `p[i]=i` initialisiert wird (`i=1,...,N`)
  - Für Vergleiche erfolgt der Zugriff auf einen Record über `a[p[i]]`
  - Vertausche Indizes `p[i]` anstatt Records

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**

- **Dann: Sortiere Verweise anstelle von Datensätzen**

- **Verfahren:**
  - Verwende ein Index-Feld `p[0..N-1]`, das mit `p[i]=i` initialisiert wird (`i=1,...,N`)
  - Für Vergleiche erfolgt der Zugriff auf einen Record über `a[p[i]]`
  - Vertausche Indizes `p[i]` anstatt Records
  - Optional: Nach dem Sortieren umsortieren der Records selbst (geht mit Aufwand $O(n)$)

© 2006-2023 Jörg Striegnitz

Indirekte Sortierverfahren

- **Bei sehr großen Datensätzen sind Bewegungen dominant für Sortieraufwand**
- **Dann: Sortiere Verweise anstelle von Datensätzen**
- **Verfahren:**
  - Verwende ein Index-Feld `p[0..N-1]`, das mit `p[i]=i` initialisiert wird (`i=1,...,N`)
  - Für Vergleiche erfolgt der Zugriff auf einen Record über `a[p[i]]`
  - Vertausche Indizes `p[i]` anstatt Records
  - Optional: Nach dem Sortieren umsortieren der Records selbst (geht mit Aufwand $O(n)$)

Jedes Sortierverfahren kann durch indirektes Sortieren so modifiziert werden, dass nicht mehr als N Record-Vertauschungen nötig sind.

© 2006-2023 Jörg Striegnitz

Umsortierung der Datensätze

- **Es gibt zwei geläufige Varianten, um die Datensätze nach einem indirekten Sortiervorgang umzusortieren:**
  - Permutation **mit zusätzlichem Feld** `b[0..N-1]`:

    `b[i]=a[p[i]]`
  - Permutation **ohne zusätzliches Feld** (*in situ*/*in place*) - lohnt nur bei wirklich großen Records
    - **Ziel**: `p[i]=i` für alle `i`
    - Falls `p[i]=i` tue nichts
    - sonst: *zyklisches Vertauschen* durchführen:
      - Kopiere Record t=a[i] - dadurch Platz (Loch an Position i)
      - „iterieren“ bis Record i an korrekte Stelle kommt; z.B.:
        t=a[2] ; a[2]=a[11] ; a[11]=a[13] ; a[13]=t
        (*siehe nächste Folien*)

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

Vor dem Sortieren:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nach dem Sortieren:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A		O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`
- Das „**S**“ ist falsch sortiert - wir merken es uns in `t`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A		O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`
- Das „**S**“ ist falsch sortiert - wir merken es uns in `t`
  Anmerkung: „**S**“ gehört an die Indexposition, 13, denn `p[13]=2`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A		O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`
- Das „**S**“ ist falsch sortiert - wir merken es uns in `t`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A		O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`
- Das „**S**“ ist falsch sortiert - wir merken es uns in `t`
- An die Position 2 gehört das **A** von Indexposition `11`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	**A**	A	O	R	T	I	N	G	E	X		M	P	L	E
p[k]	**1**		9	15	8	6	14	12	7	3	13	4	2	5	10

t
S

- Das „**A**“ ist an seiner finalen Position, denn `p[1]=1`
- Das „**S**“ ist falsch sortiert - wir merken es uns in `t`
- An die Position 2 gehört das **A** von Indexposition 11

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A		O	R	T	I	N	G	E	X		M	P	L	E
p[k]	1		9	15	8	6	14	12	7	3	13	4	2	5	10

t
S

- Das zweite „**A**“ ist an seiner finalen Position, denn `p[2]=2`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X		M	P	L	E
p[k]	1	2	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das zweite „**A**“ ist an seiner finalen Position, denn `p[2]=2`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X		M	P	L	E
p[k]	1	2	9	15	8	6	14	12	7	3	13	4	2	5	10

t: S

- Das zweite „**A**“ ist an seiner finalen Position, denn p[2]=2
- An Position 11 gehört das Das „**P**“ von Indexposition 13

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X	P	M		L	E
p[k]	1	2	9	15	8	6	14	12	7	3		4	2	5	10

t: S

- Das zweite „**A**“ ist an seiner finalen Position, denn `p[2]=2`
- An Position 11 gehört das Das „**P**“ von Indexposition 13

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X		M		L	E
p[k]	1	2	9	15	8	6	14	12	7	3		4	2	5	10

t: S

- Das „**P**“ ist an seiner finalen Position, denn `p[11]=11`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X	P	M		L	E
p[k]	1	2	9	15	8	6	14	12	7	3	11	4	2	5	10

t: S

- Das „**P**“ ist an seiner finalen Position, denn `p[11]=11`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X	P	M	S	L	E
p[k]	1	2	9	15	8	6	14	12	7	3	11	4		5	10

t

- Das „**P**“ ist an seiner finalen Position, denn `p[11]=11`
- Das „**S**“ kann jetzt an Indexposition 13 bewegt werden

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	O	R	T	I	N	G	E	X	P	M		L	E
p[k]	1	2	9	15	8	6	14	12	7	3	11	4		5	10

t

- Das „**S**“ ist an seiner finalen Position, denn `p[13]=13`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	**A**	**A**	O	R	T	I	N	G	E	X	**P**	M	**S**	L	E
p[k]	**1**	**2**	9	15	8	6	14	12	7	3	**11**	4	**13**	5	10

t

- Das „**S**“ ist an seiner finalen Position, denn `p[13]=13`

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	**A**	**A**	O	R	T	I	N	G	E	X	**P**	M	**S**	L	E
p[k]	**1**	**2**	9	15	8	6	14	12	7	3	**11**	4	**13**	5	10

t

- Das „**S**“ ist an seiner finalen Position, denn `p[13]=13`
- Es wird mit Position 3 fortgefahren ...

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	**A**	**A**	O	R	T	I	N	G	E	X	**P**	M	**S**	L	E
p[k]	**1**	**2**	9	15	8	6	14	12	7	3	**11**	4	**13**	5	10

t

- Das „**O**“ ist falsch sortiert - wir merken es uns in t
- Es wird mit Position 3 fortgefahren ...

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A		R	T	I	N	G	E	X	P	M	S	L	E
p[k]	1	2	9	15	8	6	14	12	7	3	11	4	13	5	10

t: O

- Das „**O**“ ist falsch sortiert - wir merken es uns in `t`
- Es wird mit Position 3 fortgefahren ...

© 2006-2023 Jörg Striegnitz

Beispiel: Umsortieren nach indirektem Sortieren

B *in situ* Umsortieren

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A		R	T	I	N	G	E	X	P	M	S	L	E
p[k]	1	2	9	15	8	6	14	12	7	3	11	4	13	5	10

t: O

- Das „**O**“ ist falsch sortiert - wir merken es uns in `t`
- Es wird mit Position 3 fortgefahren ...

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

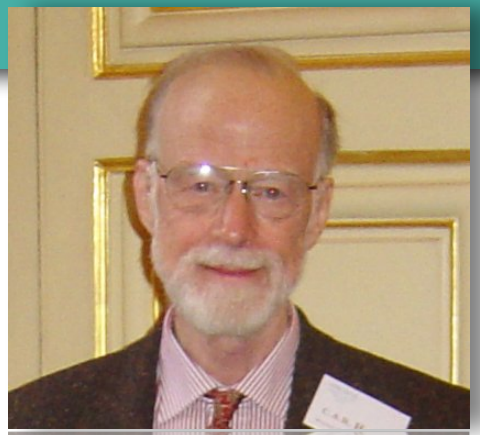
4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- **4.2.1. Quicksort**
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3.Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

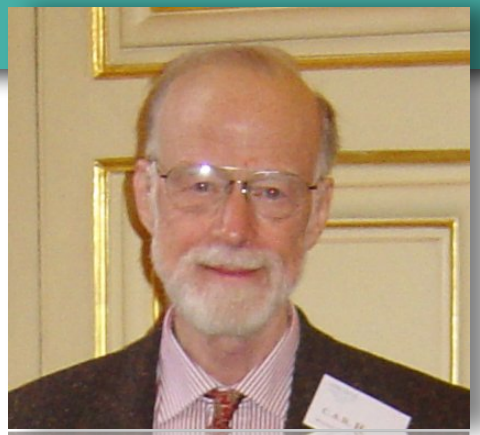
QuickSort



- **1962 von dem britischen Informatiker Charles Antony Richard Hoare veröffentlicht**

© 2006-2023 Jörg Striegnitz

QuickSort

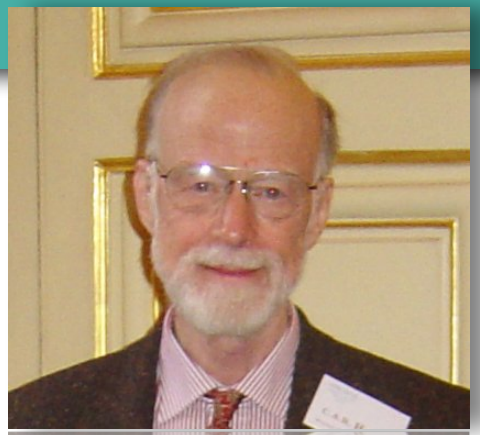


- **1962 von dem britischen Informatiker Charles Antony Richard Hoare veröffentlicht**

- **Auch bekannt als:** *Partition Exchange*-Verfahren oder „Vertauschen über große Entfernungen“

© 2006-2023 Jörg Striegnitz

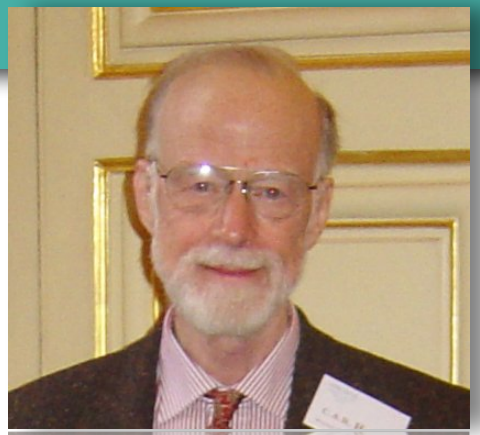
QuickSort



- **1962 von dem britischen Informatiker Charles Antony Richard Hoare veröffentlicht**
- **Auch bekannt als:** *Partition Exchange*-Verfahren oder „Vertauschen über große Entfernungen“
- **Basiert auf dem Prinzip „Teile und Herrsche“**

© 2006-2023 Jörg Striegnitz

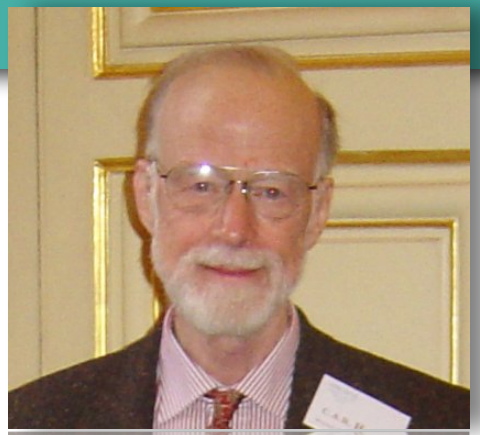
QuickSort



- **1962 von dem britischen Informatiker Charles Antony Richard Hoare veröffentlicht**
- **Auch bekannt als:** *Partition Exchange*-Verfahren oder „Vertauschen über große Entfernungen“
- **Basiert auf dem Prinzip „Teile und Herrsche“**
- **Wird oft als schnellstes internes Sortierverfahren angesehen**

© 2006-2023 Jörg Striegnitz

QuickSort



- **1962 von dem britischen Informatiker Charles Antony Richard Hoare veröffentlicht**
- **Auch bekannt als:** *Partition Exchange*-Verfahren oder „Vertauschen über große Entfernungen“
- **Basiert auf dem Prinzip „Teile und Herrsche“**
- **Wird oft als schnellstes internes Sortierverfahren angesehen**
- **Ist nicht stabil**

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

$$\forall x_1 \in F_1: \quad x_1 < \boldsymbol{p}$$
$$\forall x_2 \in F_2: \quad \boldsymbol{p} \leq x_2$$

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

$$\forall x_1 \in F_1: \quad x_1 < \boldsymbol{p}$$
$$\forall x_2 \in F_2: \quad \boldsymbol{p} \leq x_2$$

Konsequenz:
$p = \min F_2$

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

$$\forall x_1 \in F_1: \quad x_1 < \boldsymbol{p}$$
$$\forall x_2 \in F_2: \quad \boldsymbol{p} \leq x_2$$

Konsequenz:
$p = \min F_2$

(p heisst **Pivot-Element** - *pivot* (engl.)=Drehpunkt)

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

$$\forall x_1 \in F_1: \quad x_1 < \boldsymbol{p}$$
$$\forall x_2 \in F_2: \quad \boldsymbol{p} \leq x_2$$

Konsequenz:
$p = \min F_2$

(p heisst **Pivot-Element** - *pivot* (engl.)=Drehpunkt)

  - Wende dieses Schema rekursiv auf die so entstandenen Teilfolgen F_1 und F_2 an, bis diese nur noch ein Element enthalten.

© 2006-2023 Jörg Striegnitz

QuickSort - Prinzip

- **Prinzip** (Sei F eine Folge von Schlüsseln)
  - Zerlege F bezüglich eines ***partitionierenden Elements*** $p \in F$ in zwei Teilfolgen F_1 und F_2, so dass gilt:

$$\forall x_1 \in F_1: \quad x_1 < \boldsymbol{p}$$
$$\forall x_2 \in F_2: \quad \boldsymbol{p} \leq x_2$$

Konsequenz:
$p = \min F_2$

(p heisst **Pivot-Element** - *pivot* (engl.)=Drehpunkt)

  - Wende dieses Schema rekursiv auf die so entstandenen Teilfolgen F_1 und F_2 an, bis diese nur noch ein Element enthalten.
  - Setze die sortierten Teilfolgen so zu einer neuen Folge zusammen, dass p zwischen den Folgen F_1 und F_2 steht - p steht dann an seiner endgültigen Position.

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

4	11	7	1	3	2	10	6	8	9	5

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

4	11	7	1	3	2	10	6	8	9	5

1 **Wähle ein *Pivot-Element***
(hier das am weitesten rechts stehende)

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

Pivot-Element

4	11	7	1	3	2	10	6	8	9	5

1. **Wähle ein *Pivot-Element***
(hier das am weitesten rechts stehende)

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

2 **Suche von *links* nach dem ersten Element, das *größer oder gleich* dem Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

(2) **Suche von *links* nach dem ersten Element, das *größer oder gleich* dem Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

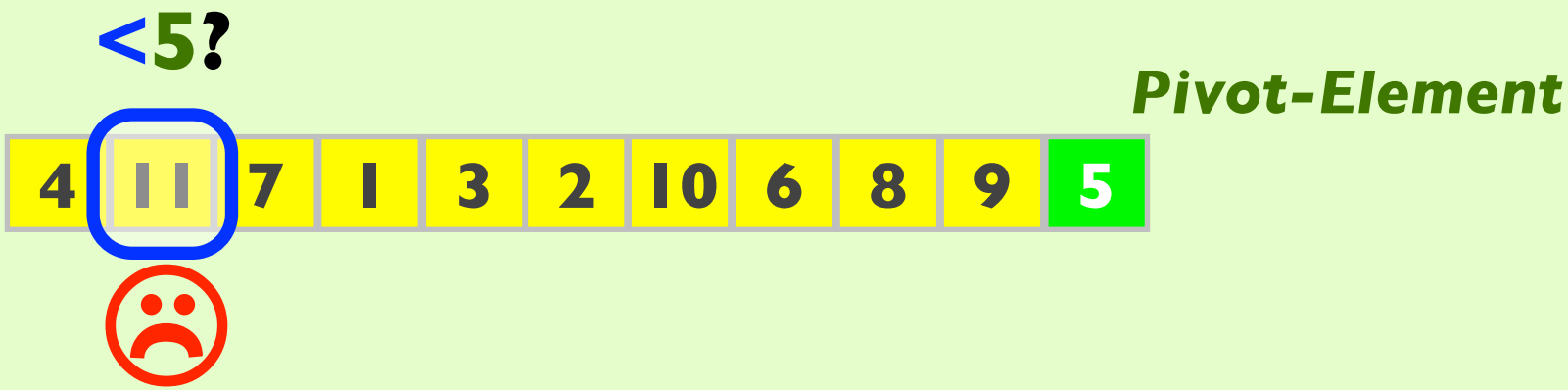
B Paritionierung bei Quicksort

2 **Suche von *links* nach dem ersten Element, das *größer oder gleich* dem Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

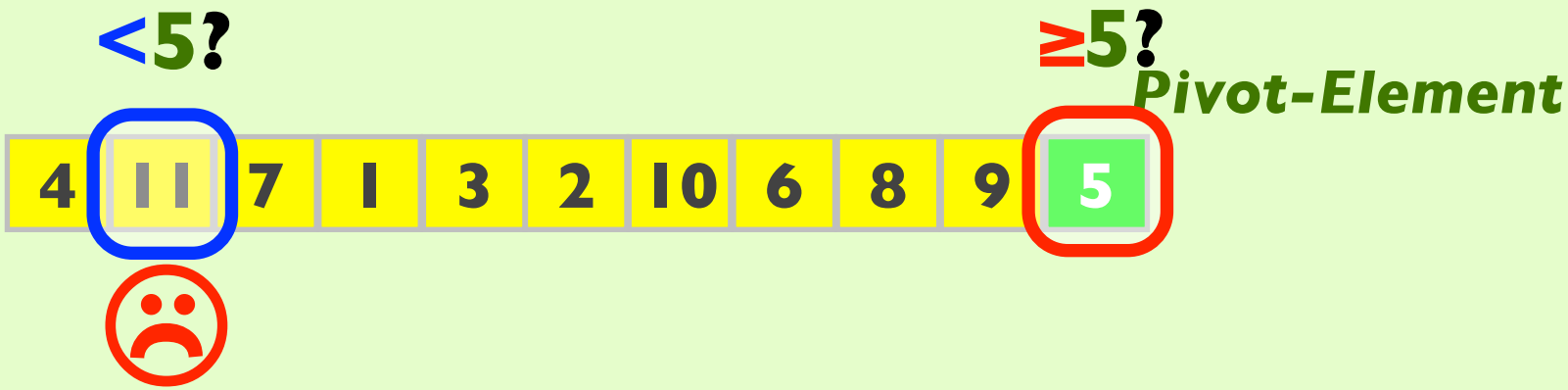


2 **Suche von *links* nach dem ersten Element, das *größer oder gleich* dem Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

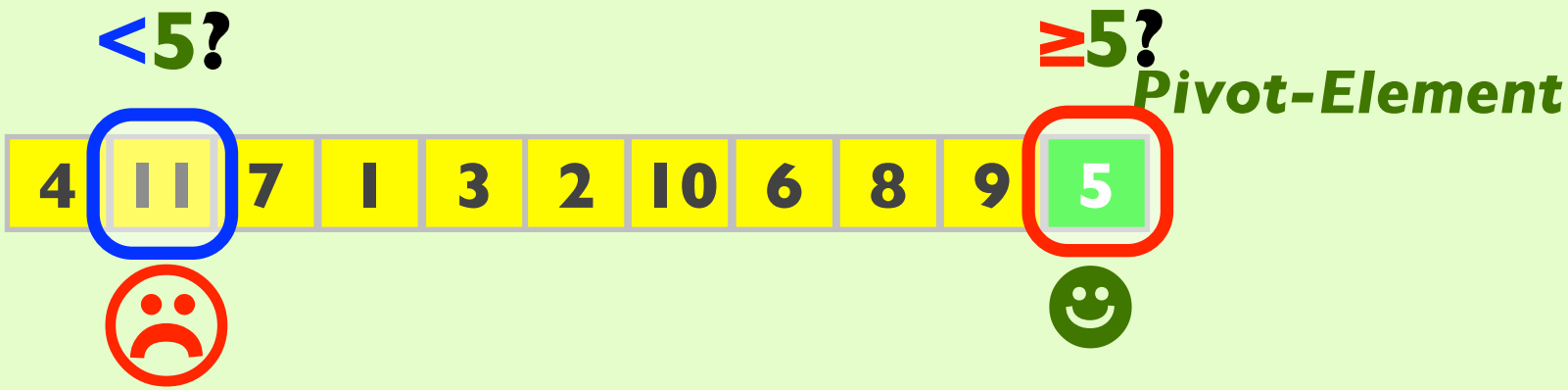


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

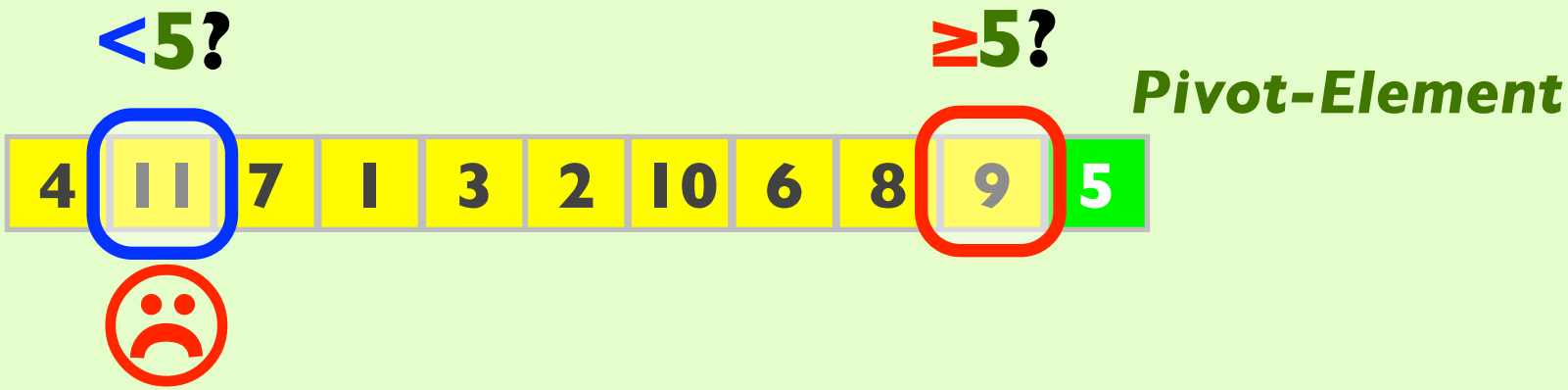


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

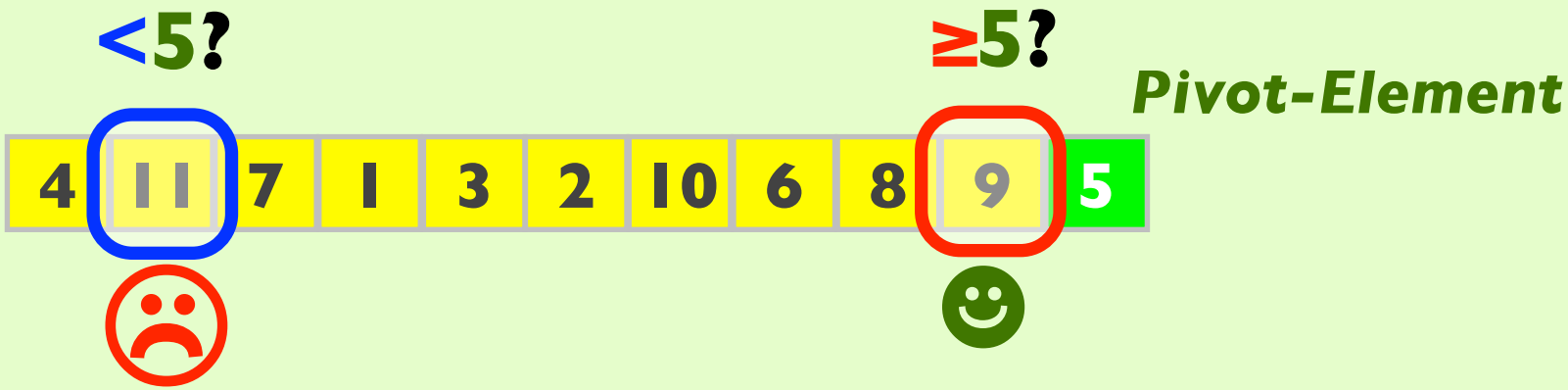


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

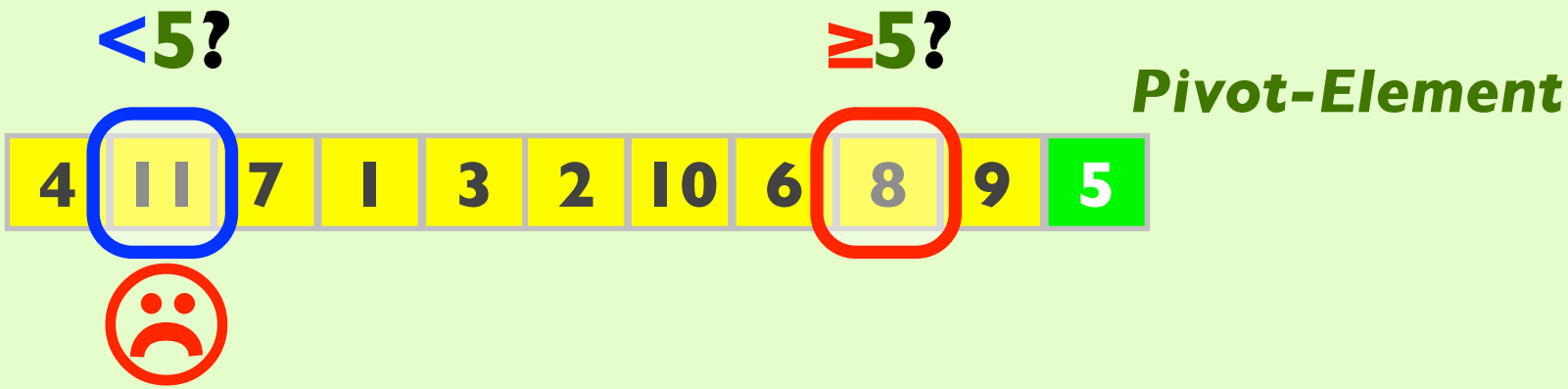


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

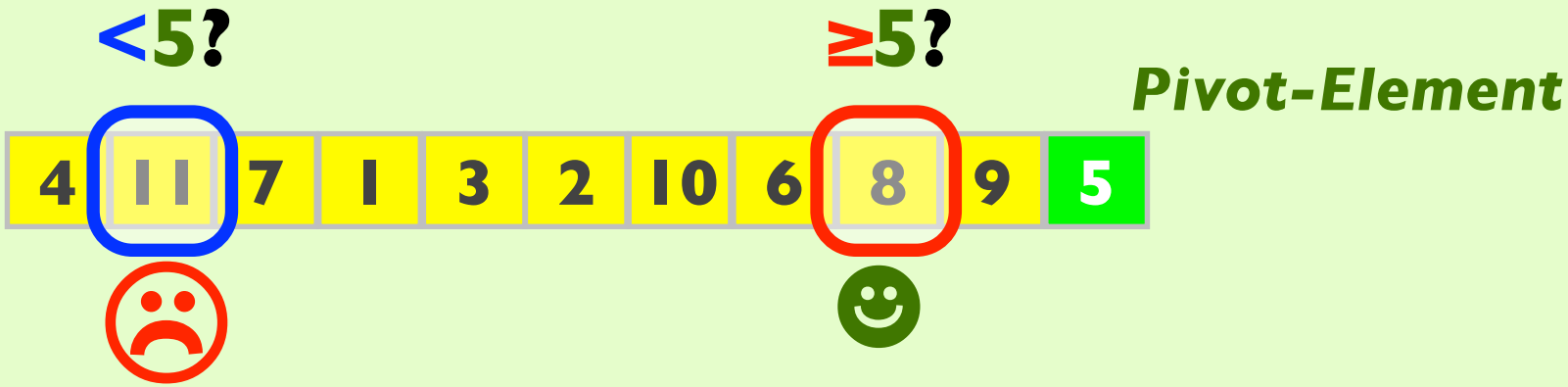


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

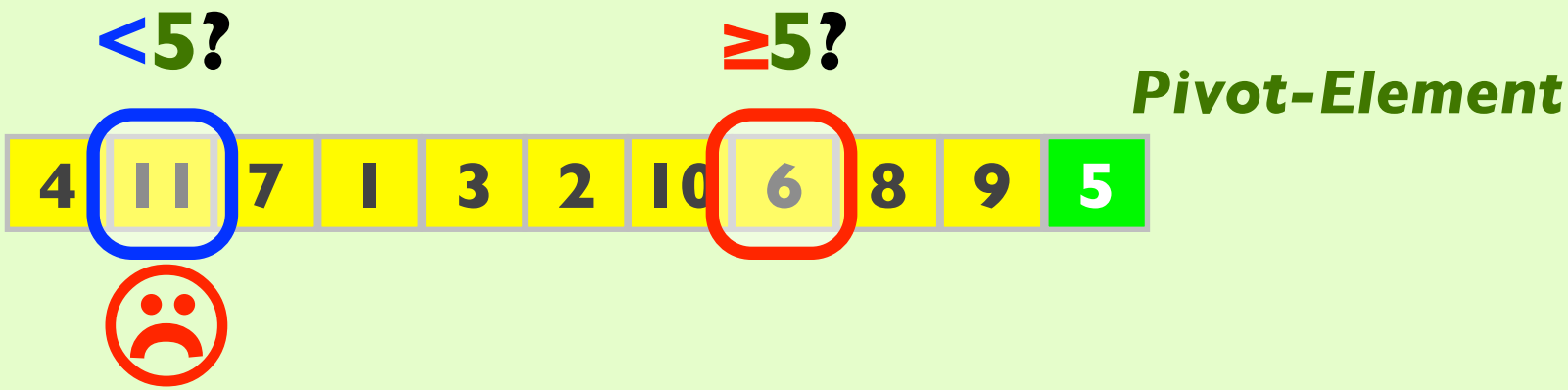


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

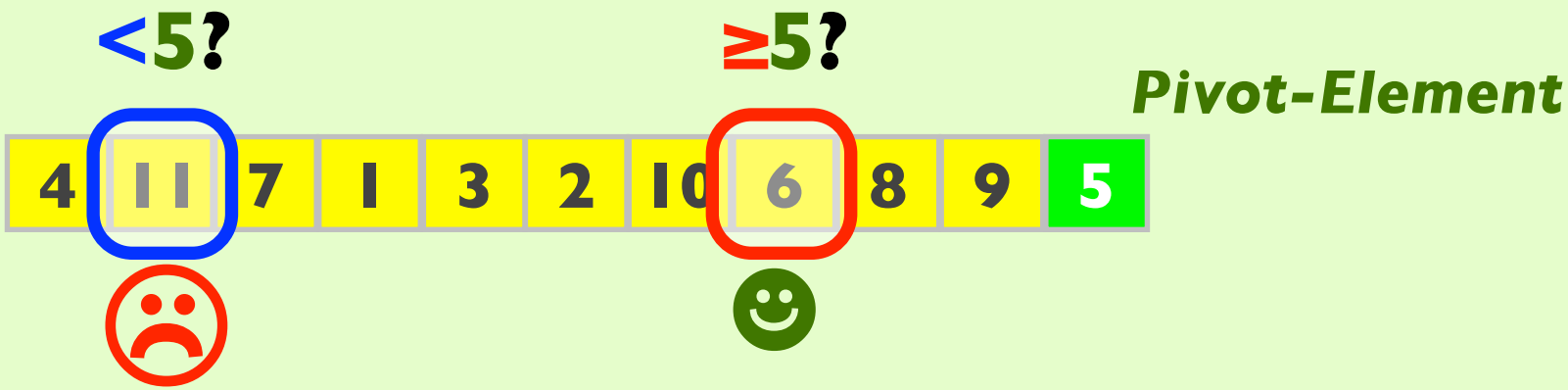


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

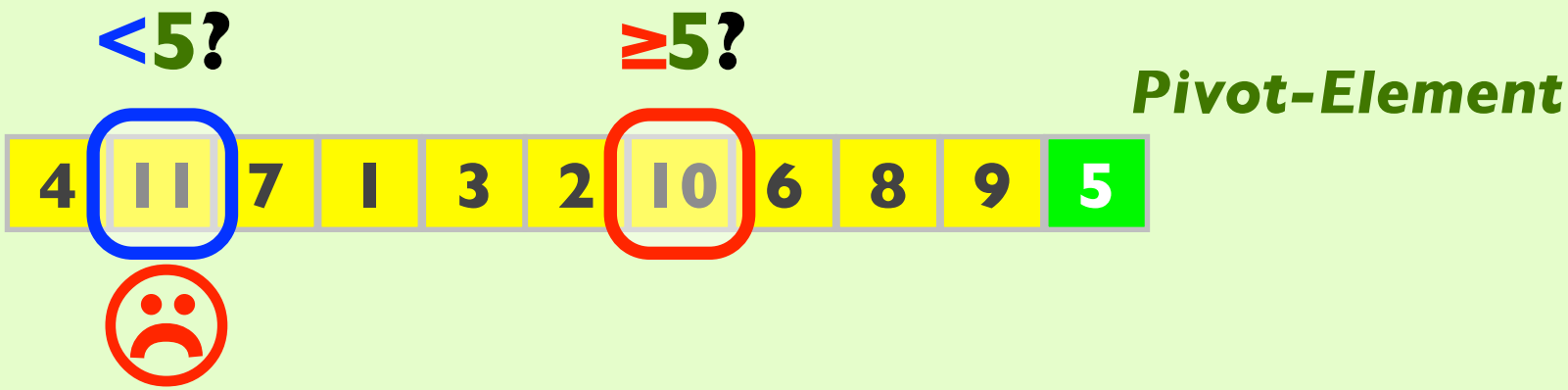


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

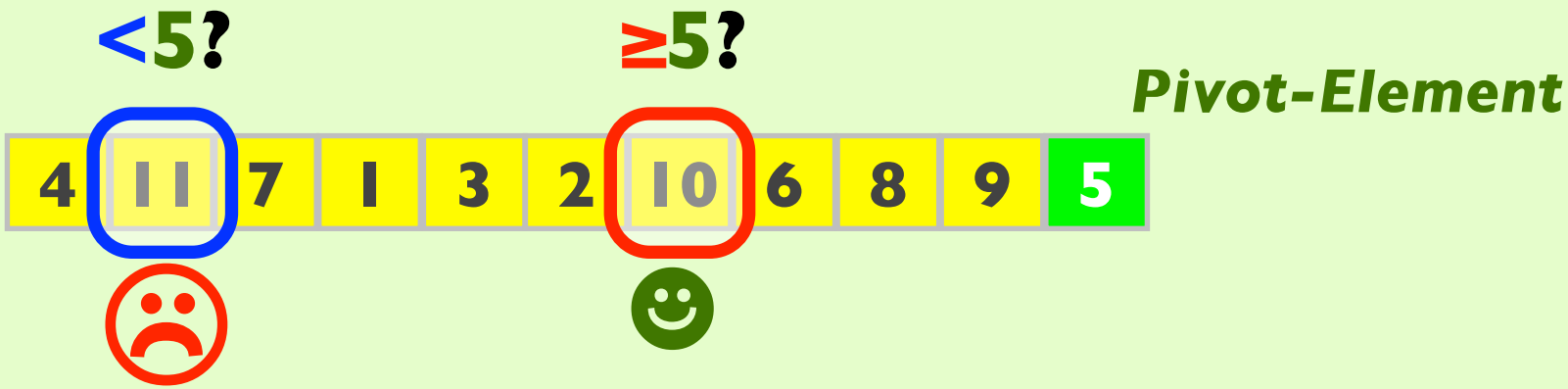


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

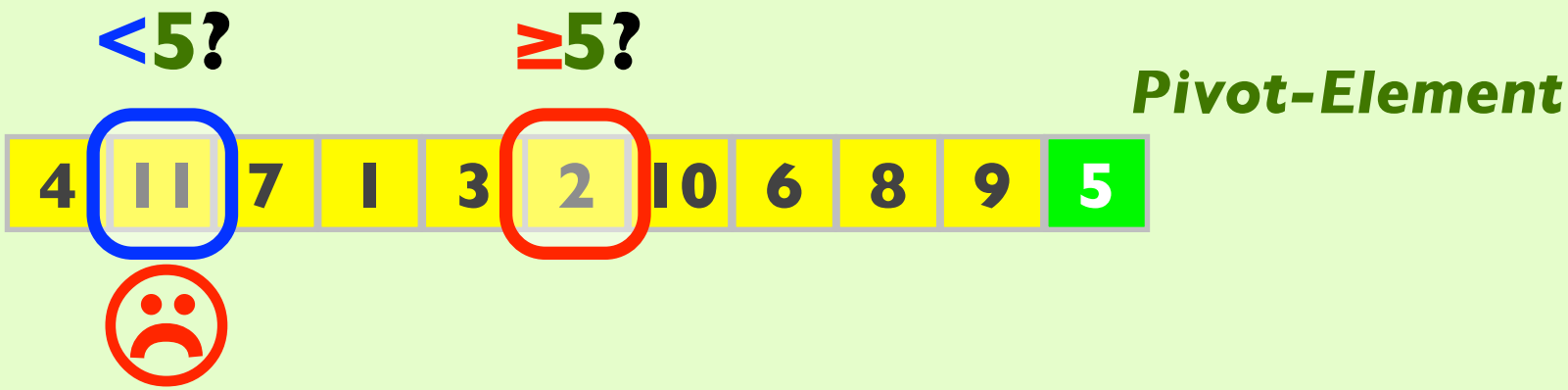


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

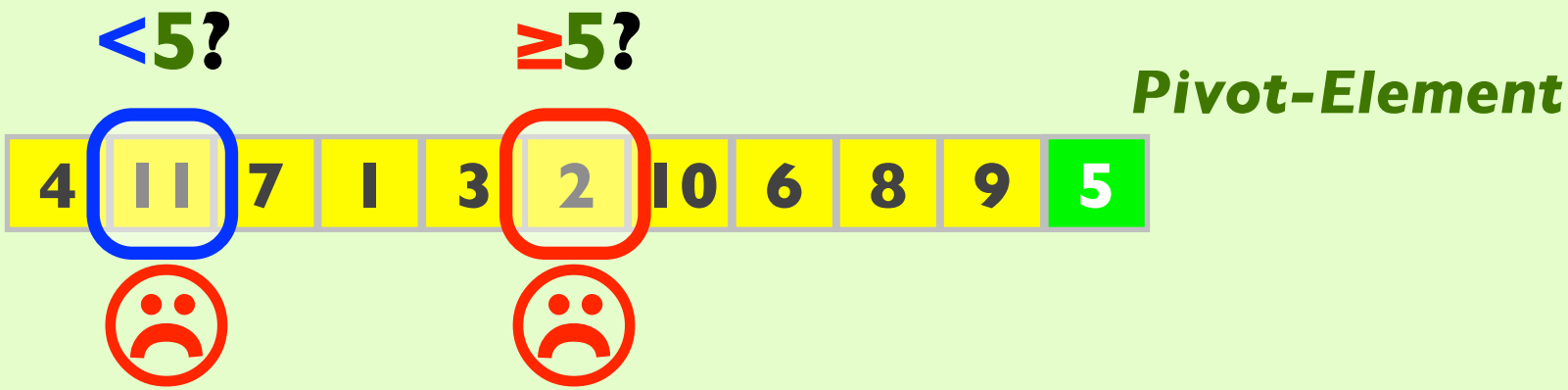


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

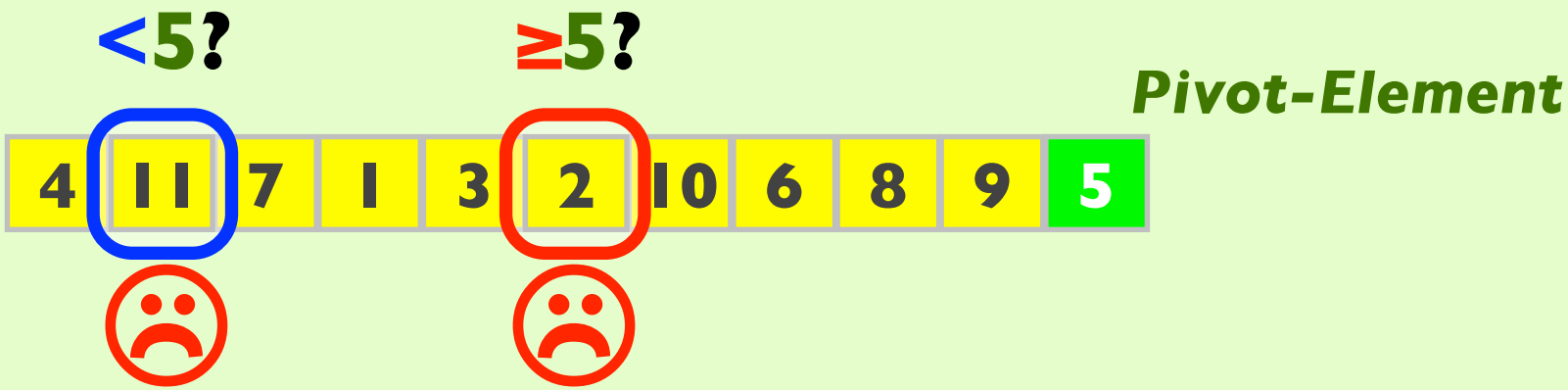


3 **Suche von *rechts* nach dem ersten Element, das *kleiner* als das Pivot-Element ist**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

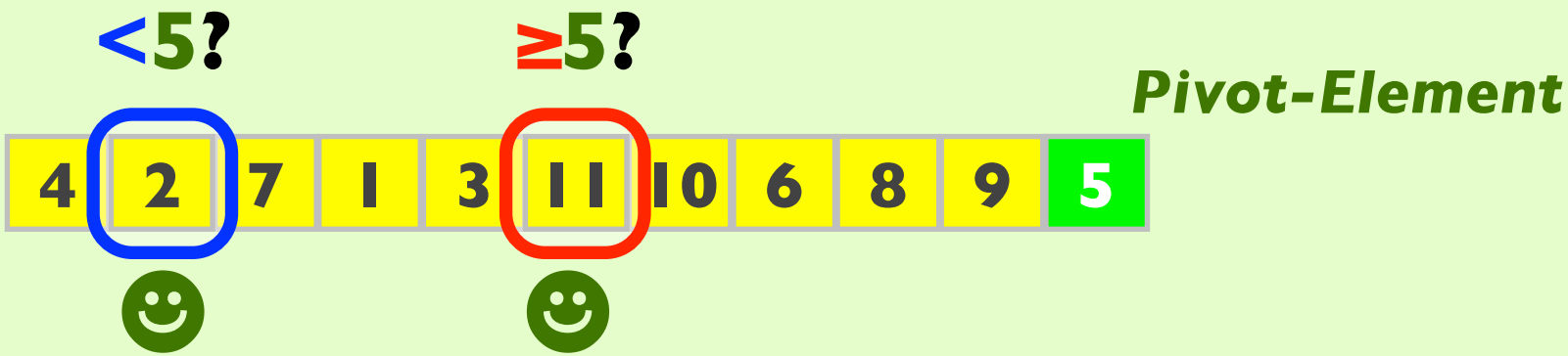


4 **Sofern entsprechende Elemente gefunden wurden, *vertausche* diese**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

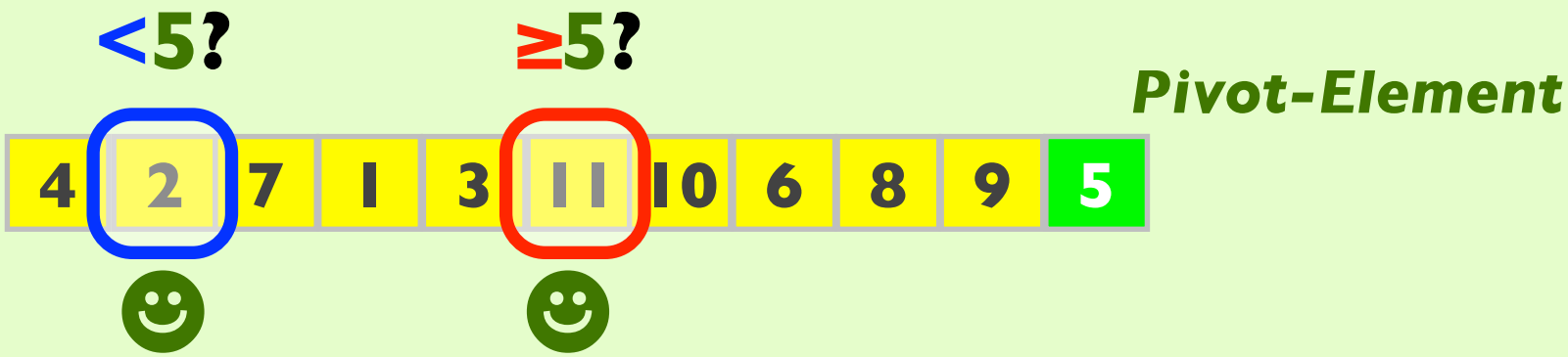


4 **Sofern entsprechende Elemente gefunden wurden, *vertausche* diese**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

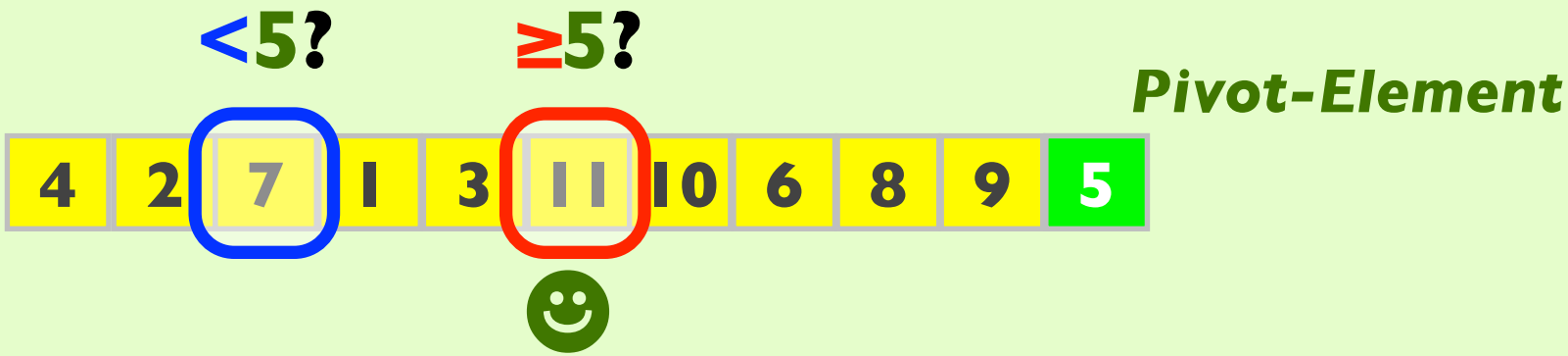


5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

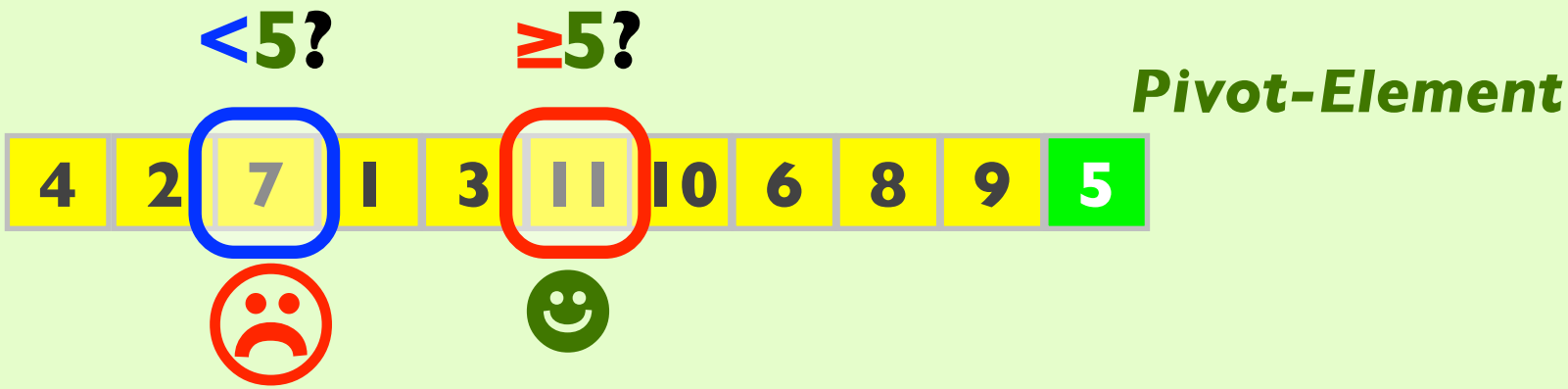


5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort



5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

5 **Fahre fort, bis sich die Indizes „kreuzen“**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

Pivot-Element

4	2	3	1	7	11	10	6	8	9	5

6 **Tausche das Pivot-Element an seine Zielposition**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

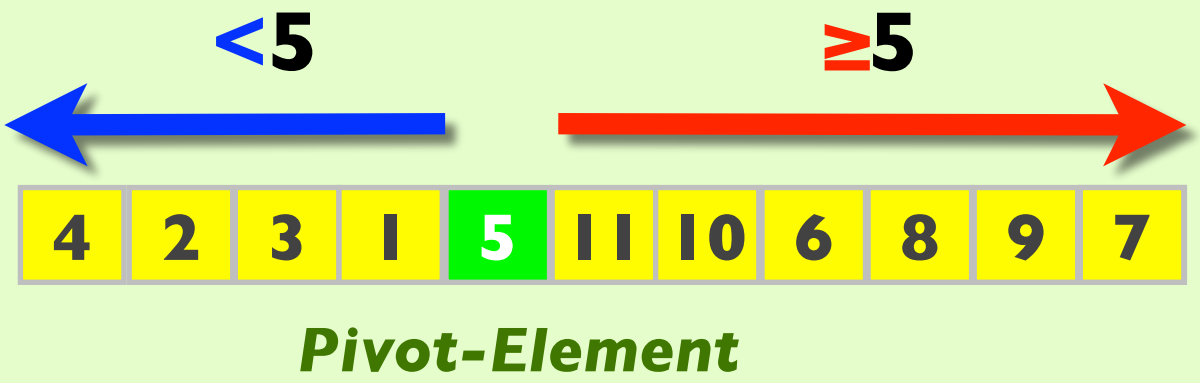
4	2	3	1	5	11	10	6	8	9	7

6 **Tausche das Pivot-Element an seine Zielposition**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort

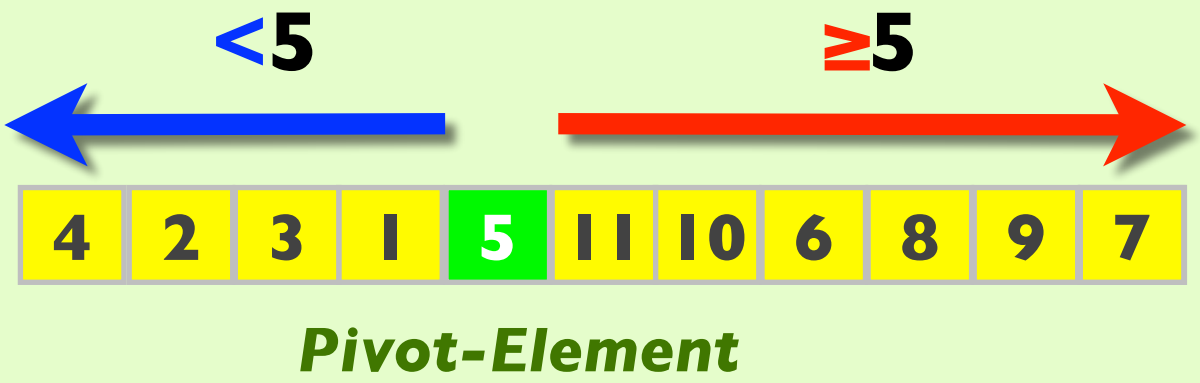


7 **Fertig, das Feld ist partitioniert!**

© 2006-2023 Jörg Striegnitz

QuickSort - Beispiel (Partitionierung)

B Paritionierung bei Quicksort



7 **Fertig, das Feld ist partitioniert!**

`Quicksort` fährt mit der Parttionierung der Teilfelder fort ...
Auf diese Weise wird schließlich das ganze Feld sortiert.

© 2006-2023 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Grundsätzlicher Aufbau von Quicksort:**

```
static void QuickSort(int l,int r,int f[]) {
  if (l<r) { // Mindestens zwei Elemente?
    // Feld Partitionieren
    int c = Partition(l,r,f);
    // ab jetzt gilt:
    //     f[l] .. f[c-1] < f[c] und
    //                      f[c] <= f[c+1] .. f[r]
    QuickSort(l, c-1, f); // linkes Feld sortieren
    QuickSort(c+1, r, f); // rechtes Feld sortieren
  }
}
```


© 2006-2023 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Wirklich interessant ist das Partitionieren:**

```
// Annahme:  Pivot-Element steht auf Position r (ggf. vorher tauschen)
// Rückgabe: Neuer Index des Pivot-Elementes
static int Partition(int l,int r,int feld[]) {
  if (l<r) {          // Gibt es was zu tun? (mindestens 2 Elemente)
    int pivot = feld[r];  // Äusserst rechtes Element ist Pivot
    int lh=l ; int rh=r;  // Indizes initialisieren
    do {
      // suche von links nach größerem Element
      while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
      // suche von rechts nach kleinerem Element
      while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
      if (lh<rh) { // nicht gekreuzt? Dann tauschen!
        int t = feld[lh] ; feld[lh]=feld[rh]; feld[rh]=t;
      }
    } while (lh<rh);
    feld[r]  = feld[lh]; feld[lh]=pivot; // Pivot-Element an Zielposition
    return lh; // Neue Pivot-Position an Aufrufer
  } else return r;
}
```


© 2006-2023 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Wirklich interessant ist das Partitionieren:**

```
// Annahme:  Pivot-Element steht auf Position r (ggf. vorher tauschen)
// Rückgabe: Neuer Index des Pivot-Elementes
static int Partition(int l,int r,int feld[]) {
  if (l<r) {          // Gibt es was zu tun? (mindestens 2 Elemente)
    int pivot = feld[r];  // Äusserst rechtes Element ist Pivot
    int lh=l ; int rh=r;  // Indizes initialisieren
    do {
      // suche von links nach größerem Element
      while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
      // suche von rechts nach kleinerem Element
      while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
      if (lh<rh) { // nicht gekreuzt? Dann tauschen!
        int t = feld[lh] ; feld[lh]=feld[rh]; feld[rh]=t;
      }
    } while (lh<rh);
    feld[r]  = feld[lh]; feld[lh]=pivot; // Pivot-Element an Zielposition
    return lh; // Neue Pivot-Position an Aufrufer
  } else return r;
}
```

Achtung! Es gibt viele Varianten der Partitionierung, die sich in der Wahl des Pivotelementes oder der Schleifenstruktur unterscheiden!
Hier: maximal n Schlüsselvergleiche!

© 2006-2023 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (Index-Vergleich)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

© 2006-2014 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (Index-Vergleich)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:

4	3	2	1	6	11	10	6	8	9	5

© 2006-2014 Jörg Striegnitz

QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (Index-Vergleich)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:

© 2006-2014 Jörg Striegnitz

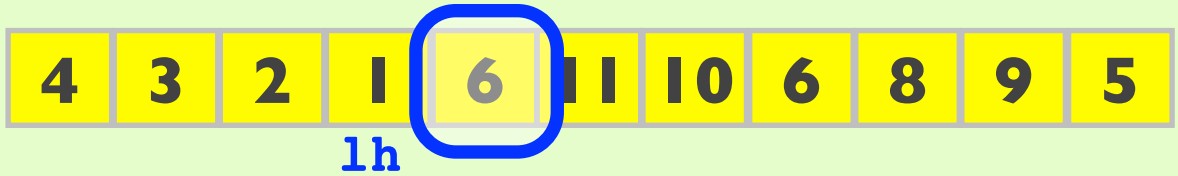
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (Index-Vergleich)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

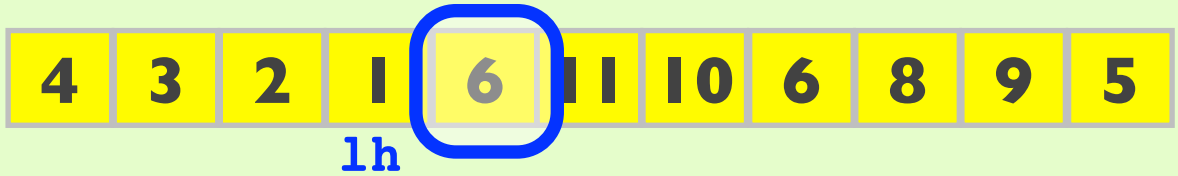
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

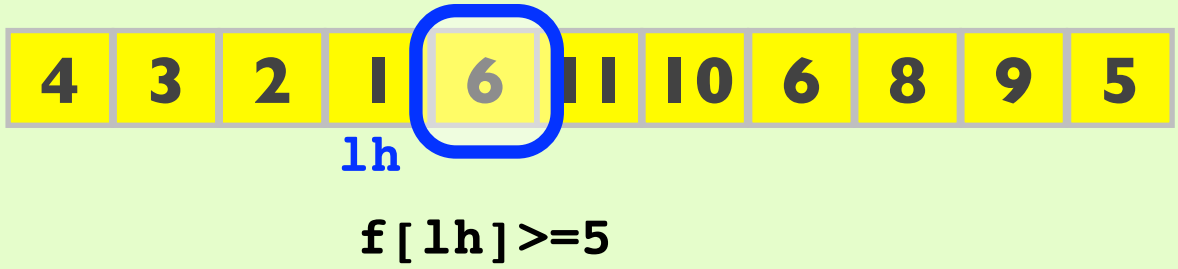
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

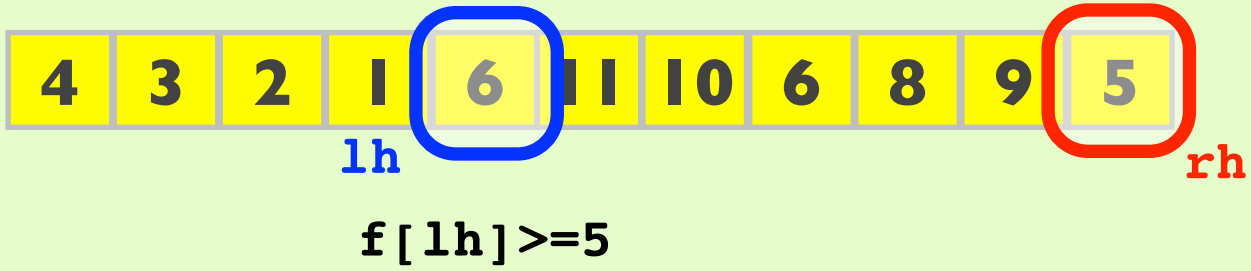
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

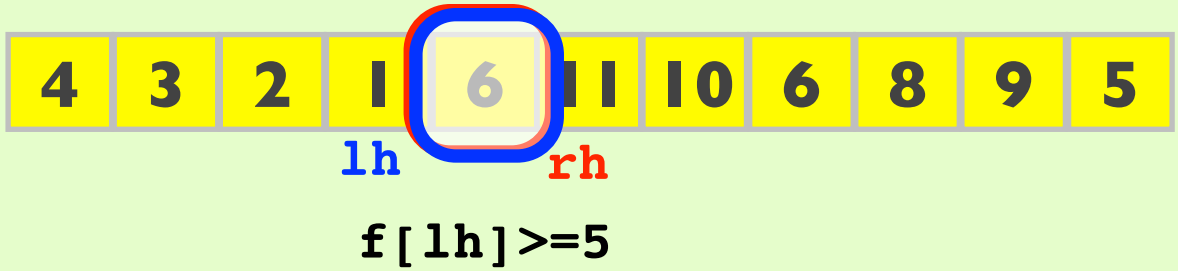
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

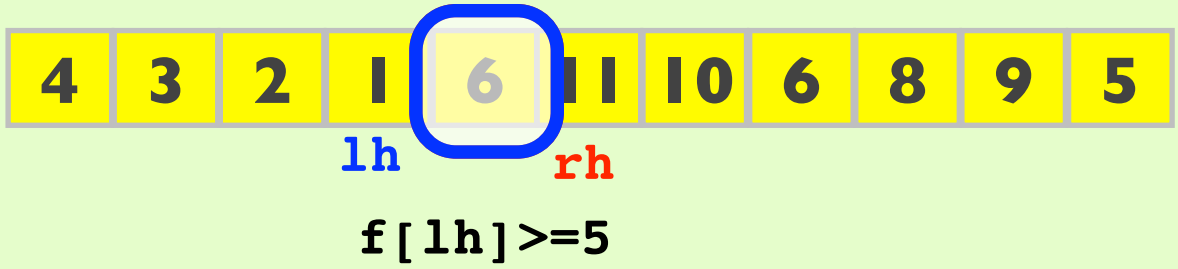
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

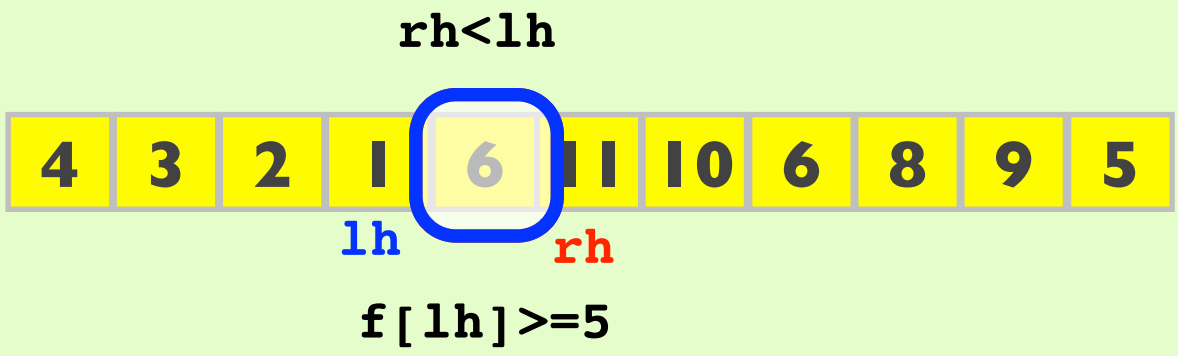
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



© 2006-2014 Jörg Striegnitz

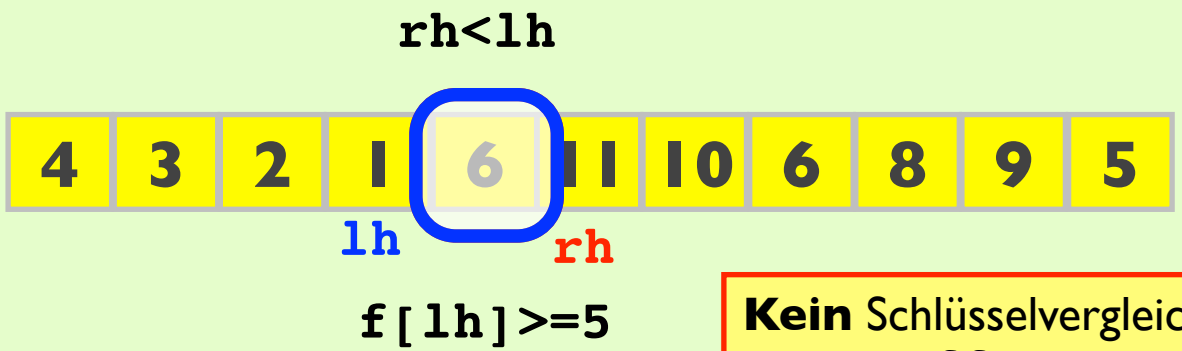
QuickSort: Implementierung (JAVA)

- **Warum maximal n Schlüsselvergleiche?**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Der optimale Fall**
  - Beide Schleifen terminieren wegen der ersten Bedingung (`lh<rh`)
  - Dann: n-1 Schlüsselvergleiche
- **Der schlechteste Fall**
  - erste terminiert wegen der zweiten Bedingung (`feld[lh] < pivot`) und
  - zweite Schleife terminiert wegen der ersten Bedingung (`rh>lh`) terminiert:

B Maximalzahl Vergleiche:



Kein Schlüsselvergleich hier, da erste Bedingung der &&-Verknüpfung schon falsch ist!
(*short-cut evaluation* von `&&`) - n **Vergleiche!**

© 2006-2014 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

4	2	3	1	5	11	10	6	8	9	7

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

4	2	3	1	5	11	10	6	8	9	7

Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

4	2	3	1	5	11	10	6	8	9	7

Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

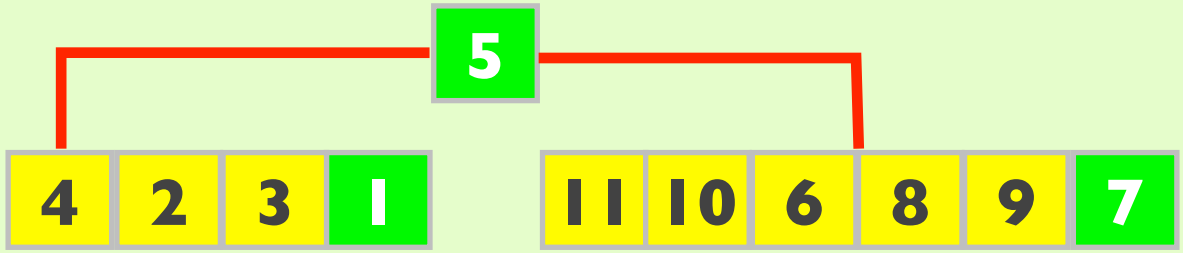
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

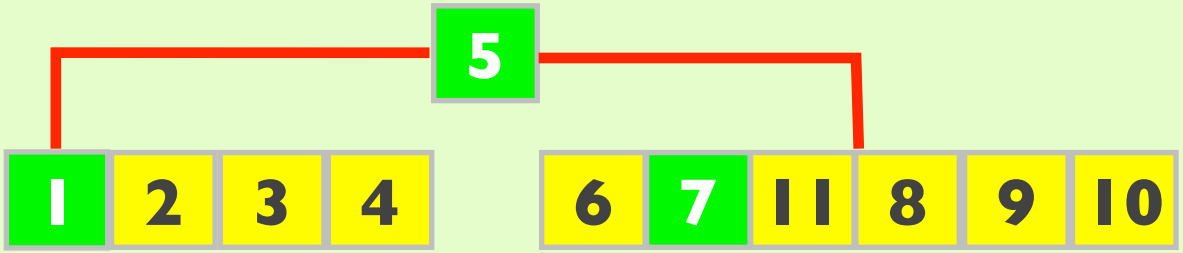
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

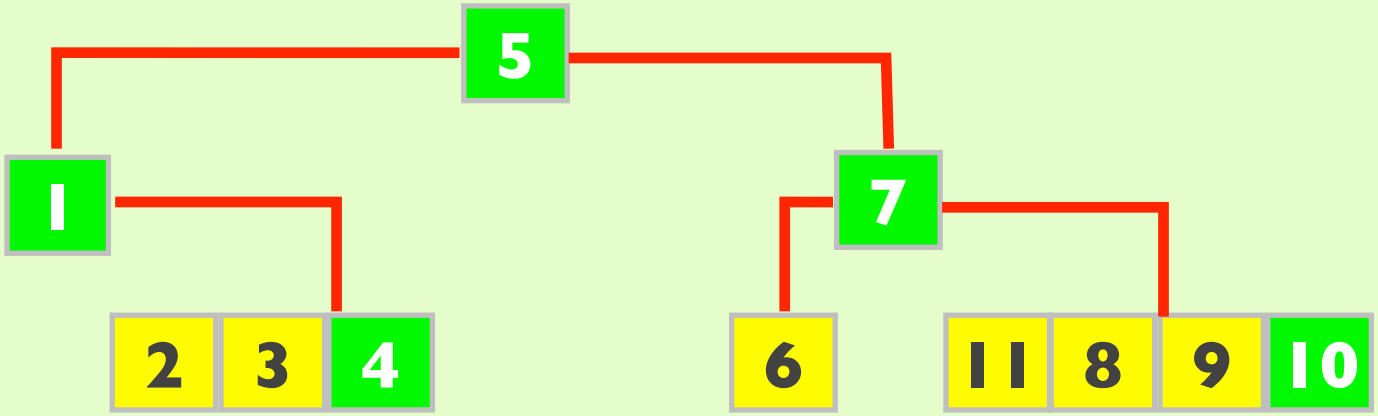
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

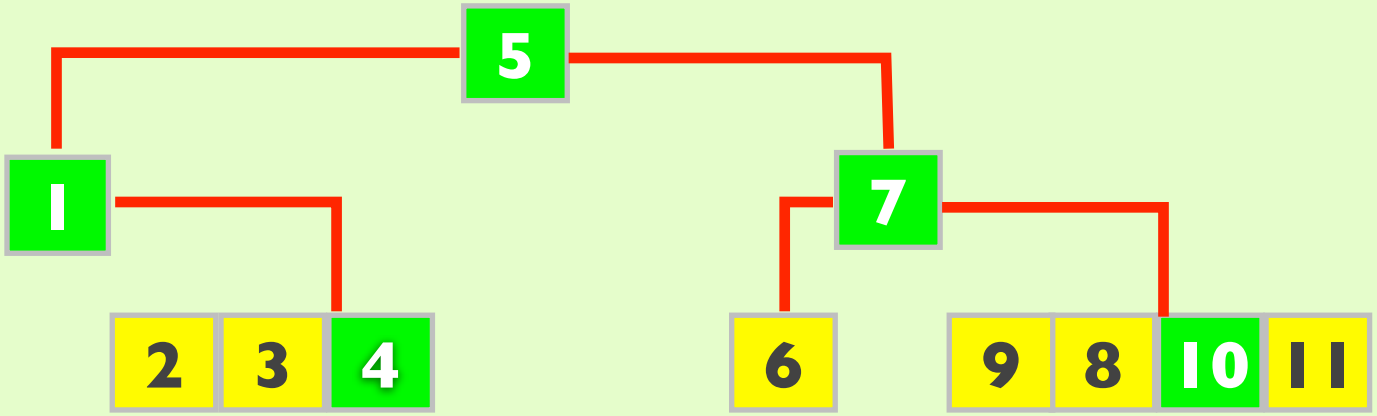
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

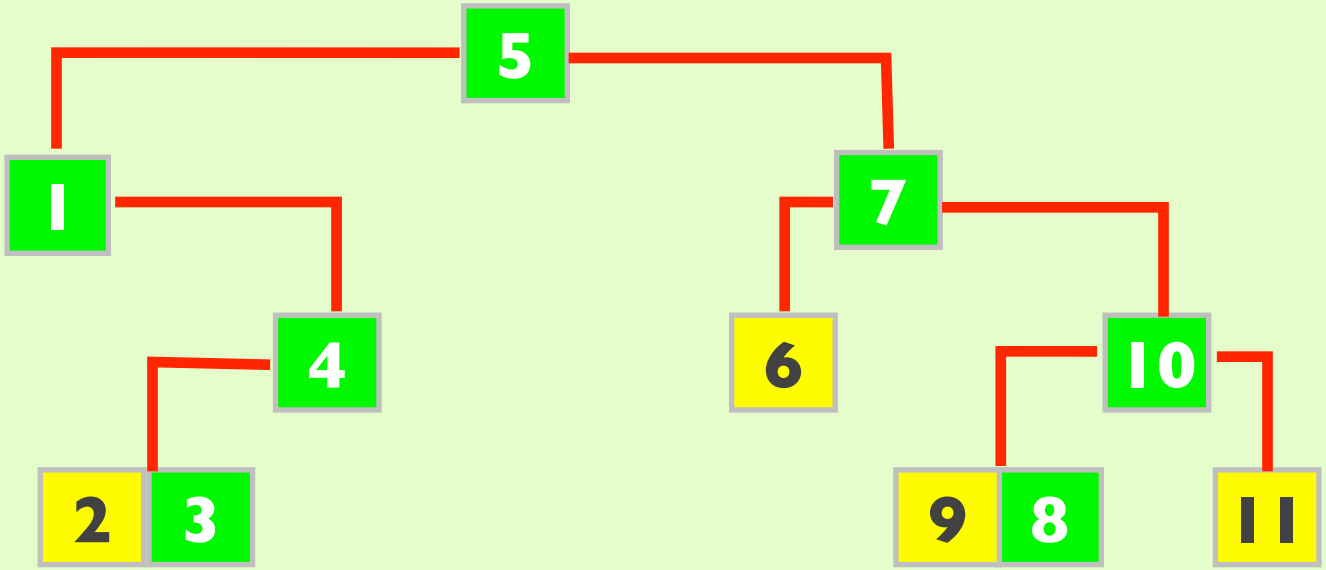
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

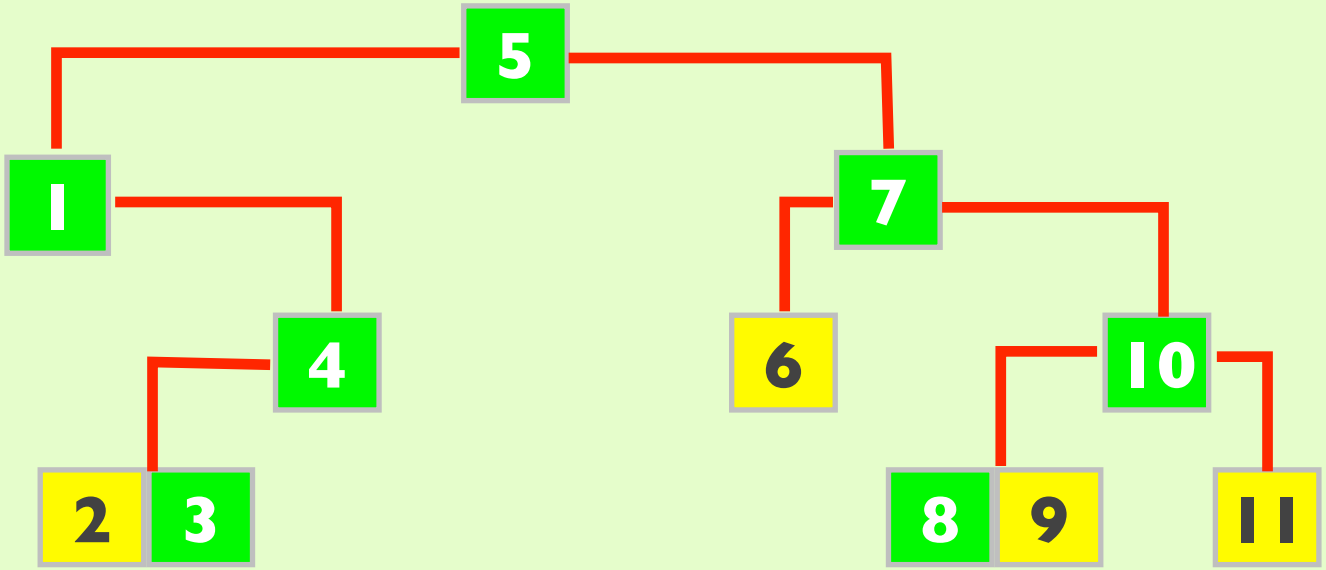
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

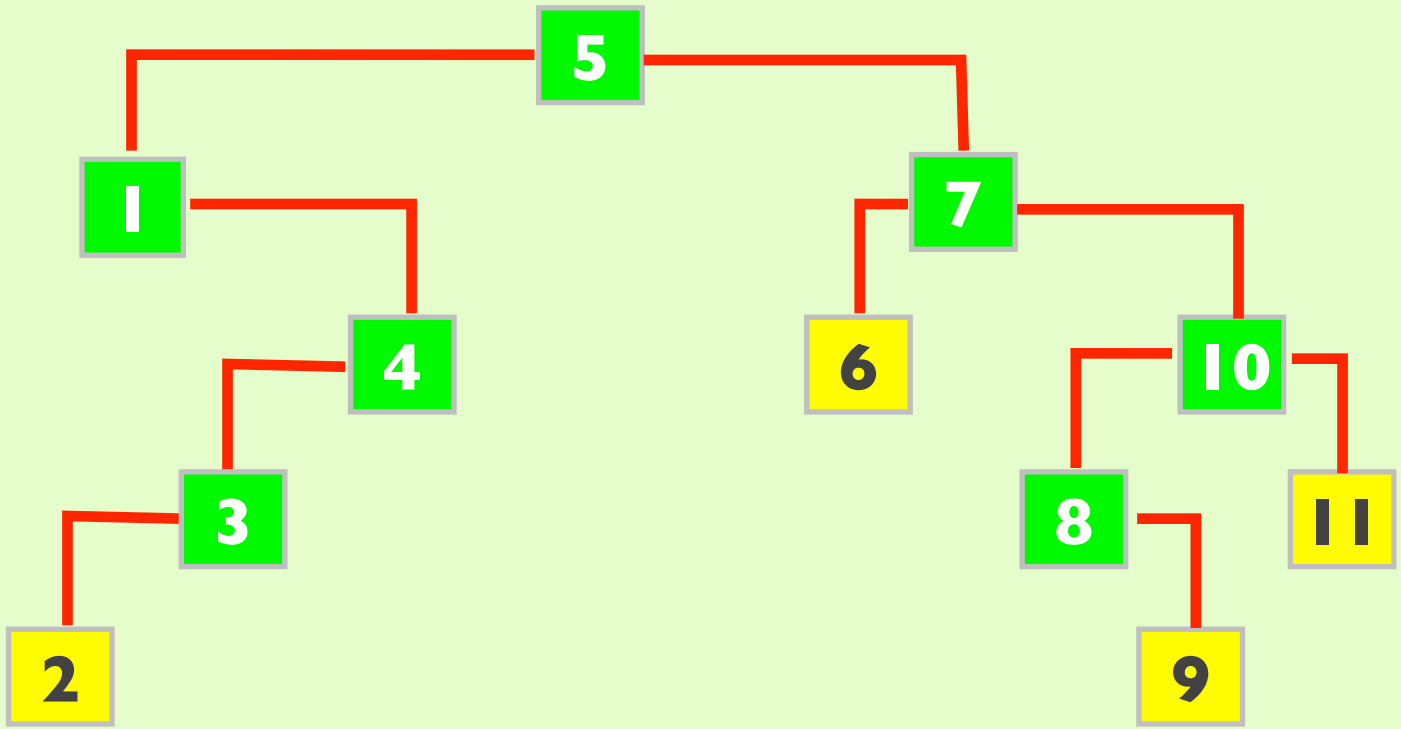
die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `Quicksort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Durch das Partitionieren wird das Feld sukzessive in zwei Hälften geteilt

die Aufrufstruktur entspricht einem Binärbaum

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

1	3	2	5	7	6	4

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

1	3	2	5	7	6	4

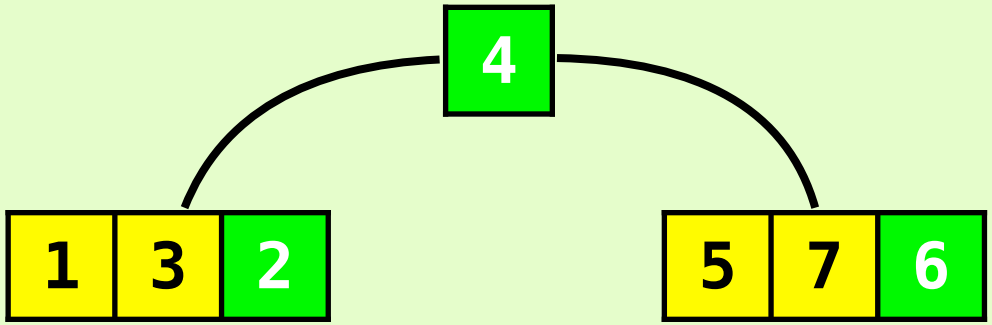
Im **günstigsten Fall** (**Pivot ist immer Median des Teilfeldes**) entsteht ein maximal gefüllter Binärbaum der Höhe $\lceil \mathrm{ld}\ n \rceil$

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



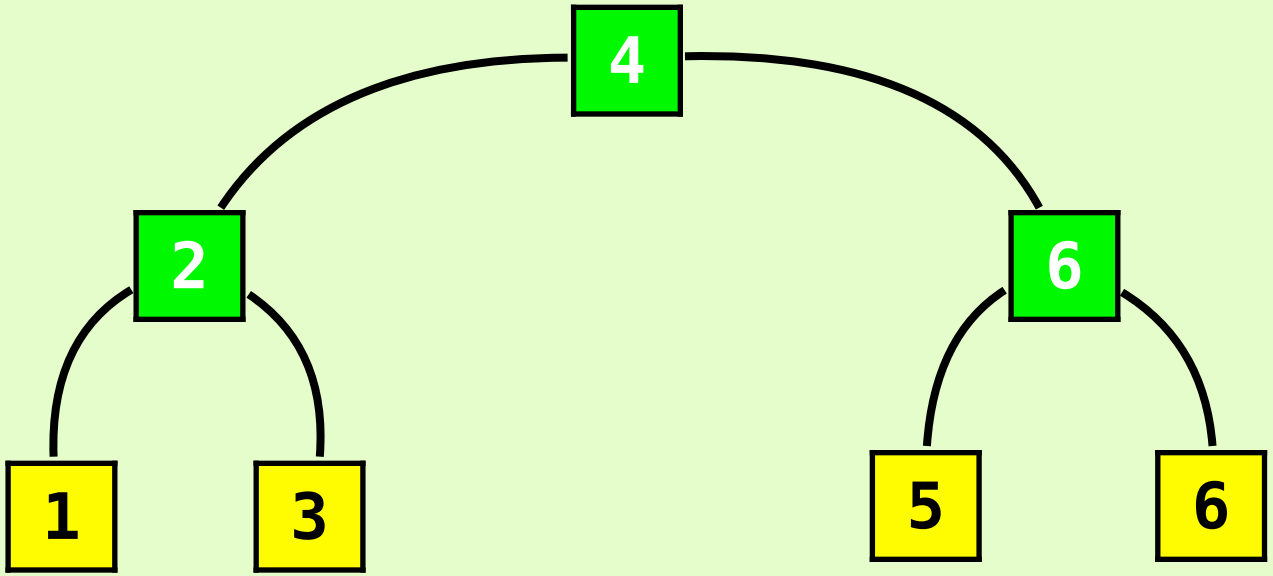
Im **günstigsten Fall** (**Pivot ist immer Median des Teilfeldes**) entsteht ein maximal gefüllter Binärbaum der Höhe $\lceil \mathrm{ld}\ n \rceil$

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Im **günstigsten Fall** (**Pivot ist immer Median des Teilfeldes**) entsteht ein maximal gefüllter Binärbaum der Höhe $\lceil \mathrm{ld}\ n \rceil$

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`

1	2	3	4	5	6	7

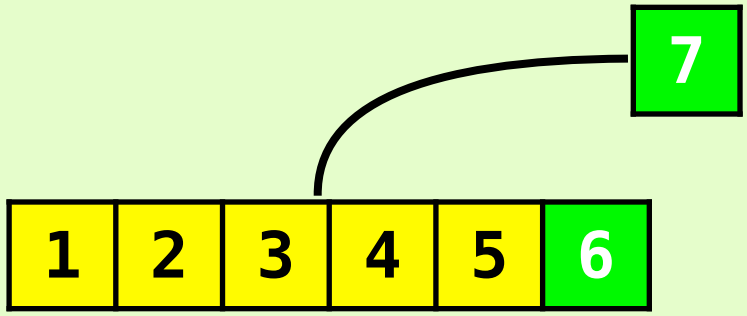
Im **ungünstigsten Fall** (**Feld ist bereits sortiert**) entsteht ein Binärbaum der Höhe n

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



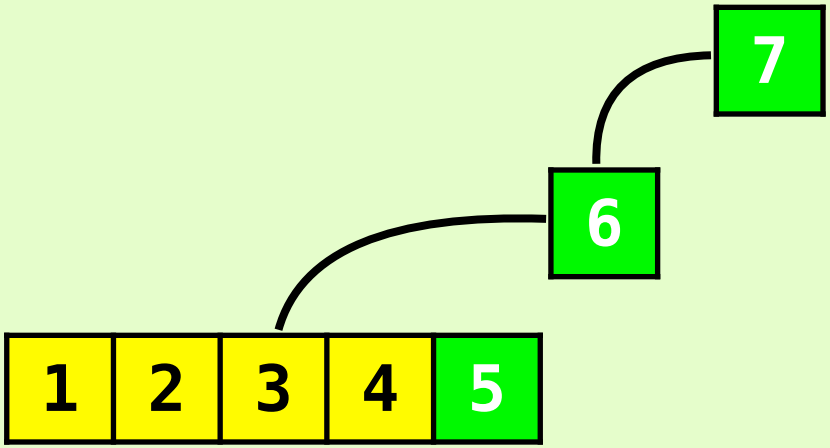
Im **ungünstigsten Fall** (**Feld ist bereits sortiert**) entsteht ein Binärbaum der Höhe n

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



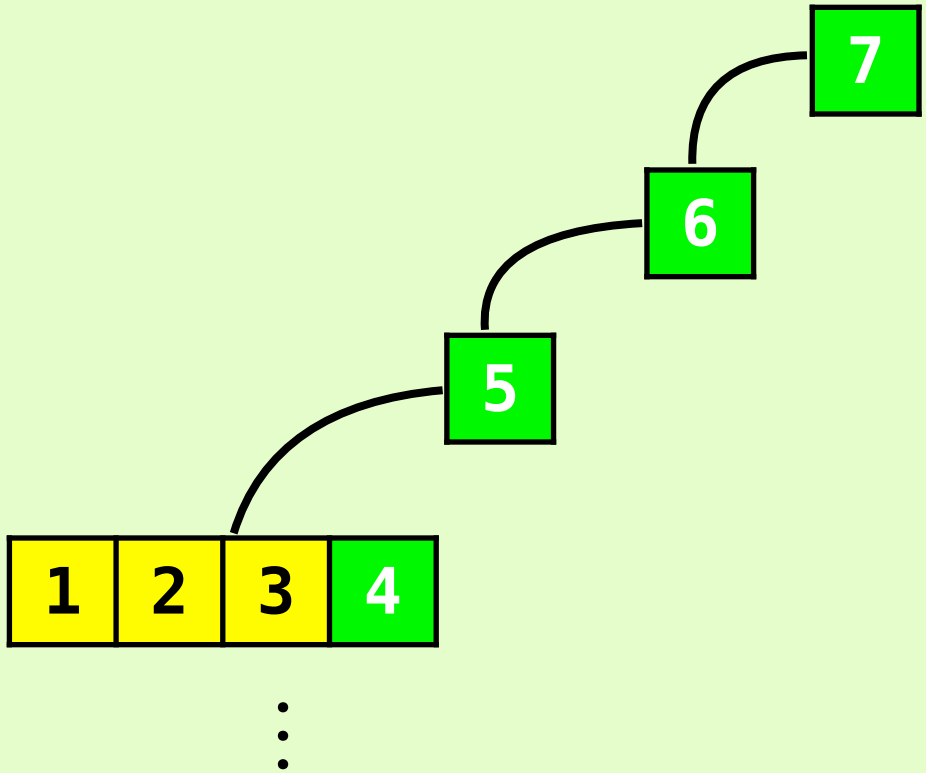
Im **ungünstigsten Fall** (**Feld ist bereits sortiert**) entsteht ein Binärbaum der Höhe n

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Wir wollen eine erste Komplexitätsabschätzung wagen ...**

B Ablauf von `Quicksort`



Im **ungünstigsten Fall** (**Feld ist bereits sortiert**) entsteht ein Binärbaum der Höhe n

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Angenommen das Pivot-Element wird in F_2 mitgeführt**
  Anmerkung: zahlreiche Implementierungen gehen so vor, da das Pivot dann nicht zwingend in F enthalten sein muss - Grund dafür folgt …

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Angenommen das Pivot-Element wird in F_2 mitgeführt**
  Anmerkung: zahlreiche Implementierungen gehen so vor, da das Pivot dann nicht zwingend in F enthalten sein muss - Grund dafür folgt …

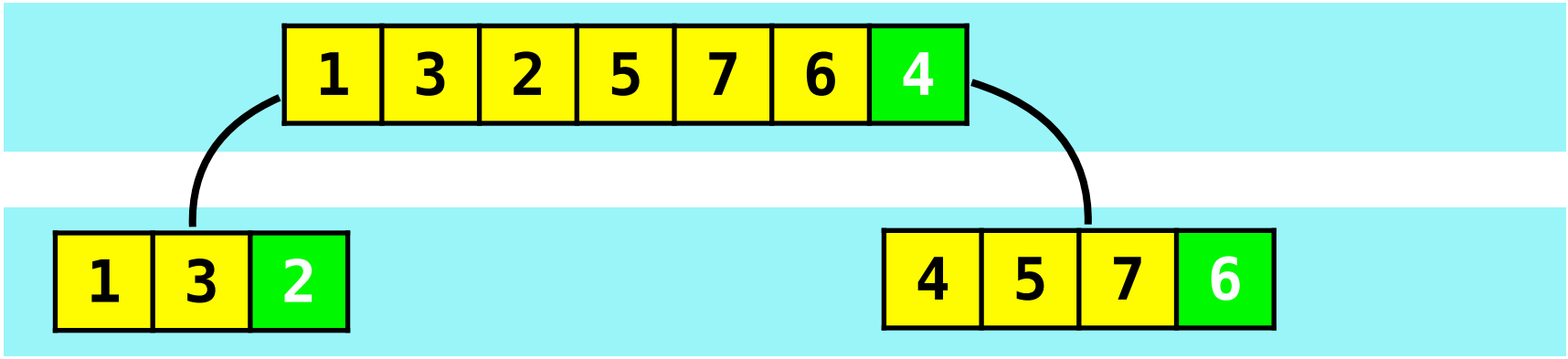
- **Dann pro Level immer n Vergleiche:**

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Angenommen das Pivot-Element wird in F_2 mitgeführt**
  Anmerkung: zahlreiche Implementierungen gehen so vor, da das Pivot dann nicht zwingend in F enthalten sein muss - Grund dafür folgt …

- **Dann pro Level immer n Vergleiche:**

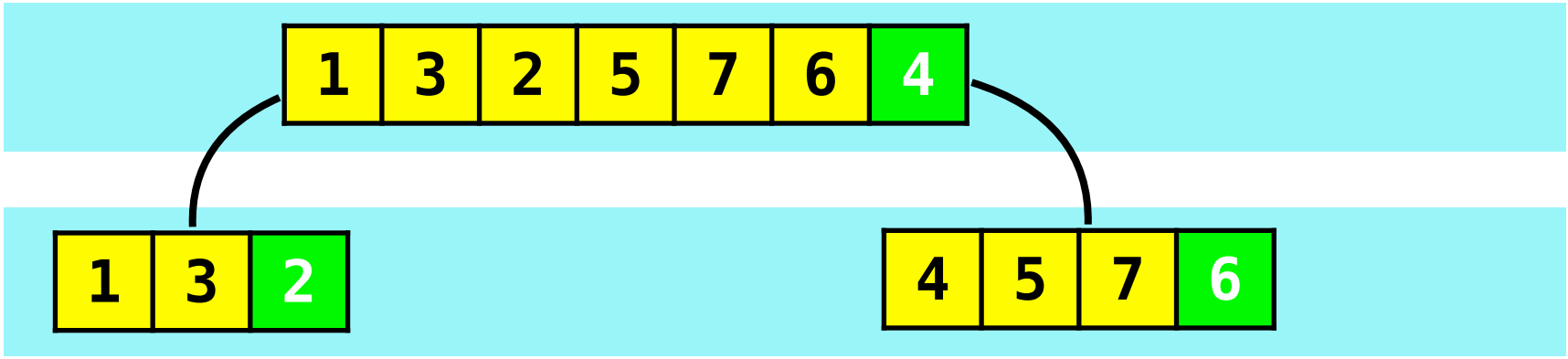


© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Angenommen das Pivot-Element wird in F_2 mitgeführt**
  Anmerkung: zahlreiche Implementierungen gehen so vor, da das Pivot dann nicht zwingend in F enthalten sein muss - Grund dafür folgt …

- **Dann pro Level immer n Vergleiche:**



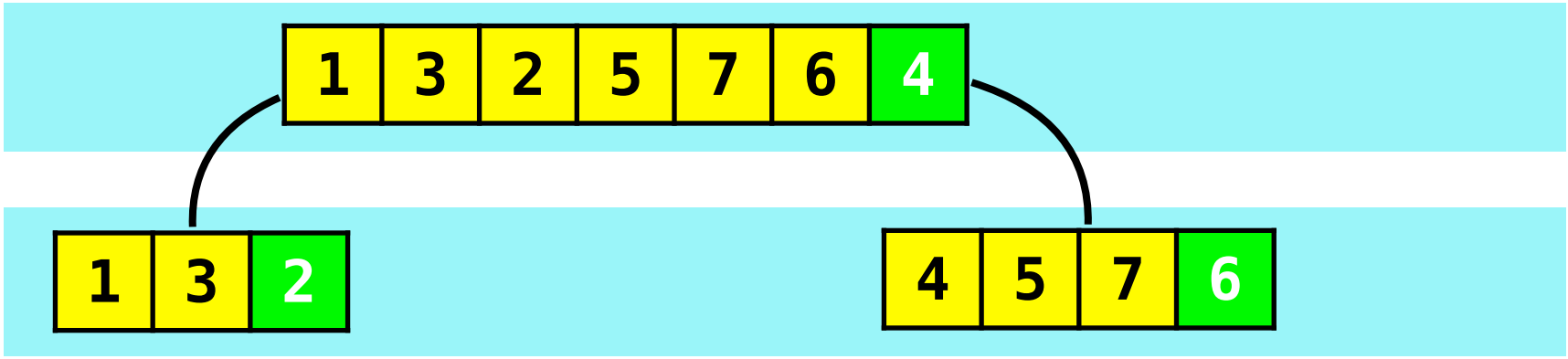
- **Somit:**
  - ➡ $\mathrm{O}(n \cdot \mathrm{ld}\ n)$ **Vergleiche im *best case***
  - ➡ $\mathrm{O}(n^2)$ **Vergleiche im *worst case***

© 2006-2023 Jörg Striegnitz

Was macht `QuickSort` schnell?

- **Angenommen das Pivot-Element wird in F_2 mitgeführt**
  Anmerkung: zahlreiche Implementierungen gehen so vor, da das Pivot dann nicht zwingend in F enthalten sein muss - Grund dafür folgt …

- **Dann pro Level immer n Vergleiche:**



- **Somit:**
  - ➡ $\mathrm{O}(n \cdot \mathrm{ld}\ n)$ **Vergleiche im *best case***
  - ➡ $\mathrm{O}(n^2)$ **Vergleiche im *worst case***

Vorsicht also bei teilweise sortierten Feldern!

© 2006-2023 Jörg Striegnitz

`QuickSort`: Komplexitätsanalyse I

- **Zahl der Vergleiche**
  - ***Best Case*** - Exakte Halbierung jedes Teil-Feldes (sei N=2^n):

© 2006-2023 Jörg Striegnitz

QuickSort: Komplexitätsanalyse I

- **Zahl der Vergleiche**
  - ***Best Case*** - Exakte Halbierung jedes Teil-Feldes (sei N=2^n):

$$
\begin{aligned}
T(N) &= N + \underbrace{\min_{1 \leq k \leq N} \{T(k-1) + T(N-k)\}}_{\text{günstigster Fall}} \\
&= N + 2 \cdot T\left(\frac{N}{2}\right)
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Komplexitätsanalyse I

- **Zahl der Vergleiche**
  - ***Best Case*** - Exakte Halbierung jedes Teil-Feldes (sei N=2^n):

Vergleiche während der Partitionierung

$$
\begin{aligned}
T(N) &= \boxed{N} + \underbrace{\min_{1 \leq k \leq N} \{T(k-1) + T(N-k)\}}_{\text{günstigster Fall}} \\
&= N + 2 \cdot T\left(\frac{N}{2}\right)
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Komplexitätsanalyse I

- **Zahl der Vergleiche**
  - ***Best Case*** - Exakte Halbierung jedes Teil-Feldes (sei N=2^n):

Vergleiche während der Partitionierung

$$
\begin{aligned}
T(N) &= \boxed{N} + \underbrace{\min_{1 \le k \le N} \{T(k-1) + T(N-k)\}}_{\text{günstigster Fall}} \\
&= N + 2 \cdot T\left(\frac{N}{2}\right)
\end{aligned}
$$

Die Lösung dieser Rekursionsgleichung hatten wir uns anschaulich schon klar gemacht (minimale Höhe eines vollständig besetzten Binärbaums):

$$T(N) = N \cdot \mathrm{ld} N$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Komplexitätsanalyse II

- **Zahl der Vergleiche**
  - ***Worst Case*** - Ungünstigste Aufteilung des Teil-Feldes
    Baum entartet zu einer linearen Liste - es wird jeweils nur ein Element „abgespalten“

© 2006-2023 Jörg Striegnitz

QuickSort: Komplexitätsanalyse II

- **Zahl der Vergleiche**
  - ***Worst Case*** - Ungünstigste Aufteilung des Teil-Feldes
    Baum entartet zu einer linearen Liste - es wird jeweils nur ein Element „abgespalten“

$$
\begin{aligned}
T(0) &= T(1) = 0 \\
T(N) &= N + \underbrace{\max_{1 \le k \le N} \{T(k-1) + T(N-k)\}}_{\text{ungünstigster Fall}} \\
& \quad (*\ \mathrm{T}(2){=}2\ *) \\
&= N + (N-1) + (N-2) + \cdots + 2 \\
&= \left(\sum_{i=1}^{N} i \right) - 1 \\
&= \frac{N \cdot (N+1)}{2} - 1
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Komplexitätsanalyse II

- **Zahl der Vergleiche**
  - ***Worst Case*** - Ungünstigste Aufteilung des Teil-Feldes
    Baum entartet zu einer linearen Liste - es wird jeweils nur ein Element „abgespalten“

$$
\begin{aligned}
T(0) &= T(1) = 0 \\
T(N) &= N + \underbrace{\max_{1 \le k \le N} \{T(k-1) + T(N-k)\}}_{\text{ungünstigster Fall}} \\
& \quad (*\ \mathrm{T}(2){=}2\ *) \\
&= N + (N-1) + (N-2) + \cdots + 2 \\
&= \left(\sum_{i=1}^{N} i \right) - 1 \\
&= \frac{N \cdot (N+1)}{2} - 1
\end{aligned}
$$

$$
T(N) \le \frac{N \cdot (N+1)}{2} - 1
$$

© 2006-2023 Jörg Striegnitz

Quicksort: Komplexitätsanalyse III

- **Zahl der Vergleiche**
  - ***Average Case*** - Wir mitteln über alle möglichen Pivot-Elemente (sei $N{=}2^n$)

$$
\begin{aligned}
T(N) &= \frac{1}{N} \cdot \sum_{k=1}^{N} [N + T(k-1) + T(N-k)] \\
&= N + \frac{1}{N} \cdot \left[\sum_{k=1}^{N} T(k-1) + \sum_{k=1}^{N} T(N-k) \right] \\
&\quad \text{(* Eigentlich sind beide Summen identisch *)} \\
&= N + \frac{1}{N} \cdot [(T(0) + \cdots + T(N-1)) + (T(N-1) + \cdots + T(0))] \\
&= N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\
&\leq 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1)
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

`Quicksort`: Komplexitätsanalyse III

- **Zahl der Vergleiche**
  - ***Average Case*** - Wir mitteln über alle möglichen Pivot-Elemente (sei $N{=}2^n$)

$$
\begin{aligned}
T(N) &= \frac{1}{N} \cdot \sum_{k=1}^{N} [N + T(k-1) + T(N-k)] \\
&= N + \frac{1}{N} \cdot \left[\sum_{k=1}^{N} T(k-1) + \sum_{k=1}^{N} T(N-k) \right] \\
&\quad \text{(* Eigentlich sind beide Summen identisch *)} \\
&= N + \frac{1}{N} \cdot [(T(0) + \cdots + T(N-1)) + (T(N-1) + \cdots + T(0))] \\
&= N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\
&\leq 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1)
\end{aligned}
$$

Auflösung später ...

© 2006-2023 Jörg Striegnitz

Nochmal Partitionierung ...

- **So partitionieren wir:**

© 2006-2023 Jörg Striegnitz

Nochmal Partitionierung ...

- **So partitionieren wir:**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```


© 2006-2023 Jörg Striegnitz

Nochmal Partitionierung ...

- **So partitionieren wir:**

```
// suche von links nach größerem Element
while ((lh<rh) && (feld[lh] < pivot)) lh=lh+1;
// suche von rechts nach kleinerem Element
while ((rh>lh) && (feld[rh] >= pivot)) rh=rh-1;
```

- **Schauen wir nochmals genau hin:**
  - Es werden nicht nur Schlüsselvergleiche vollzogen!
  - Auf die Index-Überprüfungen kann man (hier) ggf. verzichten:
    - 1. Schleife: wegen `feld[lh]< pivot` ist `pivot` **Sentinel** am Feldende
      [Sentinel = Wächter]
    - 2. Schleife: **zusätzliches** Sentinel-Element an Position `feld[-1]` mit Wert -∞

```
// suche von links nach größerem Element
while (feld[lh]<pivot) lh=lh+1; // pivot=feld[r] ist Sentinel
// suche von rechts nach kleinerem Element
while (feld[rh] >= pivot) rh=rh-1; // feld[-1] ist Sentinel
```

  - Dann: maximal $n+1$ **Schlüsselvergleiche** (Indizes „kreuzen“ sich)
    - Diese Implementierung findet man häufig in der Literatur!

© 2006-2023 Jörg Striegnitz

Exkurs

QuickSort: Average Case - *Herleitung der Anzahl der Vergleiche*

1. Fall: Partitionierung benötigt maximal n Vergleiche

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

- **Zahl der Vergleiche:** bei Partitionierung mit *n* Vergleichen
  - ***Average Case*** - Wir mitteln über alle möglichen Pivot-Elemente (sei $N{=}2^n$)

$$
\begin{aligned}
T(N) &= \frac{1}{N} \cdot \sum_{k=1}^{N} [N + T(k-1) + T(N-k)] \\
&= N + \frac{1}{N} \cdot \left[\sum_{k=1}^{N} T(k-1) + \sum_{k=1}^{N} T(N-k) \right] \\
&\quad \text{(* Eigentlich sind beide Summen identisch *)} \\
&= N + \frac{1}{N} \cdot [(T(0) + \cdots + T(N-1)) + (T(N-1) + \cdots + T(0))] \\
&= N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1)
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N)

$$T(N) = N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1)$$

Diese Gleichung ist aufgrund der Summe sehr unhandlich. Um die Summe zu eliminieren, wenden wir einen kleinen Trick an: Wenn $T(N)$ korrekt ist, dann muss auch die folgende Gleichung gelten (setze $N = N\text{-}1$)

$$T(N-1) = (N-1) + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1)$$

Jetzt stellen wir ein **Gleichungssystem** aus beiden Gleichungen auf und formen dies mit dem Ziel um, eine Gleichung für $T(N)$ ohne Summenzei–chen zu erhalten:

$$\left| \begin{array}{rcl} T(N) & = & N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\ \\ T(N-1) & = & (N-1) + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right|$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Zuerst versuchen wir identische Vorfaktoren vor den Summen zu erhalten:

$$\left|\begin{array}{rcl} T(N) & = & N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\ T(N-1) & = & (N-1) + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1) \end{array}\right| \begin{array}{l} * \, N \\ * \, N\text{-}1 \end{array}$$

$$\Leftrightarrow \left|\begin{array}{rcl} N \cdot T(N) & = & N \cdot \left(N + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1)\right) \\ (N-1) \cdot T(N-1) & = & (N-1) \cdot \left((N-1) + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1)\right) \end{array}\right|$$

$$\Leftrightarrow \left|\begin{array}{rcl} N \cdot T(N) & = & N^2 + 2 \cdot \sum_{k=1}^{N} T(k-1) \\ (N-1) \cdot T(N-1) & = & (N-1)^2 + 2 \cdot \sum_{k=1}^{N-1} T(k-1) \end{array}\right|$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

$$\left| \begin{array}{rcl} N \cdot T(N) & = & N^2 + 2 \cdot \sum_{k=1}^{N} T(k-1) \\ (N-1) \cdot T(N-1) & = & (N-1)^2 + 2 \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right|$$

Jetzt können wir die zweite von der ersten Gleichung abziehen, wodurch wir eine Gleichung ohne Summe erhalten:

$$\begin{array}{rcl} N \cdot T(N) - (N-1) \cdot T(N-1) & = & N^2 - (N-1)^2 + 2 \cdot T(N-1) \\ & = & N^2 - N^2 + 2N - 1 + 2 \cdot T(N-1) \\ & = & (2N-1) + 2 \cdot T(N-1) \end{array}$$

Durch weitere Äquivalenzumformungen gelangen wir zu:

$$\begin{array}{rrcl} & N \cdot T(N) - (N-1) \cdot T(N-1) & = & (2N-1) + 2 \cdot T(N-1) \\ \Leftrightarrow & N \cdot T(N) & = & (2N-1) + 2 \cdot T(N-1) + (N-1) \cdot T(N-1) \\ \Leftrightarrow & N \cdot T(N) & = & (2N-1) + (N+1) \cdot T(N-1) \\ \Leftrightarrow & T(N) & = & \frac{(2N-1)}{N} + \frac{(N+1) \cdot T(N-1)}{N} \end{array}$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N)

$$T(N) = \frac{(2N-1)}{N} + \frac{(N+1) \cdot T(N-1)}{N}$$

Division durch N+1 führt schließlich zu:

$$\frac{T(N)}{N+1} = \frac{(2N-1)}{N \cdot (N+1)} + \frac{T(N-1)}{N}$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N)

$$T(N) = \frac{(2N-1)}{N} + \frac{(N+1) \cdot T(N-1)}{N}$$

Division durch N+1 führt schließlich zu:

$$\boxed{\frac{T(N)}{N+1}} = \frac{(2N-1)}{N \cdot (N+1)} + \boxed{\frac{T(N-1)}{N}}$$

Wenn wir in diese beiden Terme konkrete Werte einsetzen, fällt auf, dass wir viele Quotienten doppelt generieren:

$$\frac{T(2)}{3}, \frac{T(3)}{4}, \frac{T(4)}{5}, \ldots, \frac{T(N)}{N+1} \qquad \frac{T(1)}{2}, \frac{T(2)}{3}, \frac{T(3)}{4}, \ldots, \frac{T(N-1)}{N}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

$$T(N) = \frac{(2N-1)}{N} + \frac{(N+1)\cdot T(N-1)}{N}$$

Division durch N+1 führt schließlich zu:

$$\boxed{\frac{T(N)}{N+1}} = \frac{(2N-1)}{N\cdot(N+1)} + \boxed{\frac{T(N-1)}{N}}$$

Wenn wir in diese beiden Terme konkrete Werte einsetzen, fällt auf, dass wir viele Quotienten doppelt generieren:

$$\frac{T(2)}{3}, \frac{T(3)}{4}, \frac{T(4)}{5}, \ldots, \frac{T(N)}{N+1} \qquad \frac{T(1)}{2}, \frac{T(2)}{3}, \frac{T(3)}{4}, \ldots, \frac{T(N-1)}{N}$$

Wir erzeugen jetzt ein großes Gleichungssystem, indem wir für N Werte von N bis 2 einsetzen. Wenn wir diese Gleichungen dann addieren, fallen die doppelten Terme raus.

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Wir stellen ein Gleichungssystem aus N-2 Gleichungen auf:

$$
\left|
\begin{array}{ccccc}
\frac{T(N)}{N+1} & = & \frac{T(N-1)}{N} & + & \frac{2N-1}{N\cdot(N+1)} \\
\frac{T(N-1)}{N} & = & \frac{T(N-2)}{N-1} & + & \frac{(2(N-1)-1)}{(N-1)\cdot N} \\
\frac{T(N-2)}{N-1} & = & \frac{T(N-3)}{N-2} & + & \frac{(2(N-2)-1)}{(N-2)\cdot(N-1)} \\
 & \vdots & & & \\
\frac{T(3)}{4} & = & \frac{T(2)}{3} & + & \frac{2\cdot 3-1}{2\cdot 4} \\
\frac{T(2)}{3} & = & \frac{T(1)}{2} & + & \frac{3}{2\cdot 3}
\end{array}
\right.
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Wir stellen ein Gleichungssystem aus N-2 Gleichungen auf:

$$
\left|
\begin{array}{ccccc}
\frac{T(N)}{N+1} & = & \frac{T(N-1)}{N} & + & \frac{2N-1}{N\cdot(N+1)} \\
\frac{T(N-1)}{N} & = & \frac{T(N-2)}{N-1} & + & \frac{(2(N-1)-1)}{(N-1)\cdot N} \\
\frac{T(N-2)}{N-1} & = & \frac{T(N-3)}{N-2} & + & \frac{(2(N-2)-1)}{(N-2)\cdot(N-1)} \\
 & \vdots & & & \\
\frac{T(3)}{4} & = & \frac{T(2)}{3} & + & \frac{2\cdot 3-1}{2\cdot 4} \\
\frac{T(2)}{3} & = & \frac{T(1)}{2} & + & \frac{3}{2\cdot 3}
\end{array}
\right.
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Wir stellen ein Gleichungssystem aus N-2 Gleichungen auf:

$$
\begin{array}{rcl}
\frac{T(N)}{N+1} & = & \frac{T(N-1)}{N} + \frac{2N-1}{N\cdot(N+1)} \\
\frac{T(N-1)}{N} & = & \frac{T(N-2)}{N-1} + \frac{(2(N-1)-1)}{(N-1)\cdot N} \\
\frac{T(N-2)}{N-1} & = & \frac{T(N-3)}{N-2} + \frac{(2(N-2)-1)}{(N-2)\cdot(N-1)} \\
 & \vdots & \\
\frac{T(3)}{4} & = & \frac{T(2)}{3} + \frac{2\cdot 3-1}{2\cdot 4} \\
\frac{T(2)}{3} & = & \frac{T(1)}{2} + \frac{3}{2\cdot 3}
\end{array}
$$

Durch Addition aller Gleichungen fallen die **rot** markierten Terme raus. Wir erhalten:

$$\frac{T(N)}{N+1} = \frac{2N-1}{N\cdot(N+1)} + \frac{2(N-1)-1}{N\cdot N} + \cdots + \frac{2\cdot 2-1}{2\cdot(2+1)} + \frac{T(1)}{2}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Jetzt folgt Fleissarbeit ...

$$\frac{T(N)}{N+1} = \frac{2N-1}{N\cdot(N+1)} + \frac{2(N-1)-1}{(N-1)\cdot N} + \cdots + \frac{2\cdot 2-1}{2\cdot(2+1)} + \frac{T(1)}{2}$$

(* Wir "isolieren" die störende Eins ... *)

$$= \frac{2N}{N\cdot(N+1)} - \frac{1}{N\cdot(N+1)} + \frac{2(N-1)}{(N-1)\cdot N} - \frac{1}{(N-1)\cdot N} + \cdots + \frac{2\cdot 2}{2(2+1)} - \frac{1}{2(2+1)} + \frac{T(1)}{2}$$

(* ... kürzen ... *)

$$= \frac{2}{(N+1)} - \frac{1}{N\cdot(N+1)} + \frac{2}{N} - \frac{1}{(N-1)\cdot N} + \cdots + \frac{2}{(2+1)} - \frac{1}{2(2+1)} + \frac{T(1)}{2}$$

(* ... und fassen ähnliche Brüche in Summen zusammen:*)

$$= 2\cdot\sum_{i=3}^{N+1}\frac{1}{i} - \sum_{i=2}^{N}\frac{1}{i\cdot(i+1)} + \frac{T(1)}{2}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Durch eine einfache Umformung kommen wir zu einer interessanten Summe:

$$\frac{T(N)}{N+1} = 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \sum_{i=2}^{N} \frac{1}{i \cdot (i+1)} + \frac{T(1)}{2}$$

(* Durch einfache Indexverschiebung können wir die zweite Summe *)

(* als negative fallende Faktorielle schreiben: *)

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \sum_{i=1}^{N-1} \frac{1}{(i+1) \cdot (i+2)} + \frac{T(1)}{2}$$

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \sum_{i=1}^{N-1} i^{\underline{-2}} + \frac{T(1)}{2}$$

Summen über fallende Faktorielle lassen sich sehr gut handhaben, da man für diese sehr leicht eine äquivalente Teleskopreihe findet, die man dann kollabieren kann [*siehe: Concrete Mathematics S47 ff bzw. nächste Folie*]. . Hier schonmal die Teleskopreihe und ihr Kollaps:

$$\sum_{i=1}^{N-1} \frac{1}{(i+1) \cdot (i+2)} = \sum_{i=1}^{N-1} \frac{1}{(i+1)} - \frac{1}{(i+2)}$$

$$= \frac{1}{1+1} - \frac{1}{1+2} + \frac{1}{2+1} - \frac{1}{2+2} + \cdots + \frac{1}{(N-1)+1} - \frac{1}{(N-1)+2}$$

$$= \frac{1}{2} - \frac{1}{N+1} = \frac{(N-1)}{2 \cdot (N+1)}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Jetzt folgt wieder Fleissarbeit ...

$$\frac{T(N)}{N+1} = 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \sum_{i=1}^{N-1} i^{\underline{-2}} + \frac{T(1)}{2}$$

(* Jetzt können wir die zweite Summe auflösen: *)

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \left. \frac{i^{\underline{-1}}}{-1} \right|_{1}^{(N-1)+1} + \frac{T(1)}{2}$$

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \left(\frac{N^{\underline{-1}}}{-1} - \frac{1^{\underline{-1}}}{-1} \right) + \frac{T(1)}{2}$$

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \left(\frac{\frac{1}{(N+1)}}{-1} - \frac{\frac{1}{2}}{-1} \right) + \frac{T(1)}{2}$$

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \left(-\frac{1}{(N+1)} + \frac{1}{2} \right) + \frac{T(1)}{2}$$

$$= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \frac{(N-1)}{2 \cdot (N+1)} + \frac{T(1)}{2}$$

© 2006-2023 Jörg Striegnitz

Anmerkungen zur fallenden Faktoriellen

- **Es ist** $x^{\underline{m}} = x \cdot (x-1) \cdot (x-2) \cdot \dots \cdot (x-m+1) = \prod_{i=0}^{m-1} (x-i)$
- **Ferner gilt** $\frac{x^{\underline{m}}}{x^{\underline{m-1}}} = x - m + 1$ und $x^{\underline{0}=1}$
- **also:** $\frac{x^{\underline{3}}}{x^{\underline{2}}} = x - 2 \qquad \frac{x^{\underline{2}}}{x^{\underline{1}}} = x - 1 \qquad \frac{x^{\underline{1}}}{x^{\underline{0}}} = x - 0$
- **logische Fortsetzung für negative** m**:** $\frac{x^{\underline{0}}}{x^{\underline{-1}}} = x - (-1) = \frac{1}{x^{\underline{-1}}}$

$$x^{\underline{-1}} = \frac{1}{(x+1)}$$

nun ist

$$\frac{x^{\underline{-1}}}{x^{\underline{-2}}} = x + 2$$

daraus folgt

$$x^{\underline{-2}} = \frac{x^{\underline{-1}}}{(x+2)} = \frac{1}{(x+1) \cdot (x+2))}$$

also allgemein:

$$x^{\underline{-m}} = \frac{1}{(x+1)(x+2)\cdots(x+m)}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

$$\begin{aligned}\frac{T(N)}{N+1} &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \frac{(N-1)}{2 \cdot (N+1)} + \frac{T(1)}{2} \\ &\quad \text{(* Mit T(1)=0 folgt sofort: *)} \\ &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \frac{(N-1)}{2 \cdot (N+1)}\end{aligned}$$

Jetzt stört nur noch die **harmonische Reihe**, welche man wie folgt abschätzen kann

$$\sum_{i=1}^{N} \frac{1}{i} \approx \ln N + \gamma \quad (\gamma \text{ ist die Euler-Mascheroni-Konstante ; } \gamma = 0.57721\ldots)$$

Da unsere Summe nicht bei 1 startet, schätzen wir erneut ab:

$$\begin{aligned}\sum_{k=m}^{N+1} \frac{1}{k} &\geq \ln(N+1) - \underbrace{\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{m-1}\right)}_{\text{(* Harmonische Reihe ... *)}} \\ &\geq \ln(N+1) - \ln(m-1) \\ &\geq \ln \frac{N+1}{m-1}\end{aligned}$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N)

Aus $\displaystyle\sum_{k=m}^{N+1} \frac{1}{k} \geq \ln \frac{N+1}{m-1}$ und $\displaystyle\sum_{k=m-1}^{N+1} \frac{1}{k} \geq \sum_{k=m}^{N+1} \frac{1}{k}$

folgt sofort

$$\ln \frac{N+1}{m-1} \leq \sum_{k=m}^{N+1} \frac{1}{k} \leq \ln \frac{N+1}{m-2}$$

Setzen wir diese Erkenntnis um, erhalten wir

$$\begin{aligned}
\frac{T(N)}{N+1} &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} - \frac{(N-1)}{2 \cdot (N+1)} \\
&\leq 2 \cdot \ln \frac{N+1}{3-2} - \frac{(N-1)}{2 \cdot (N+1)} \\
&\leq 2 \cdot \ln(N+1) - \frac{(N-1)}{2 \cdot (N+1)}
\end{aligned}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Setzen wir diese Erkenntnis um, erhalten wir

$$
\begin{array}{rrcl}
 & \frac{T(N)}{N+1} & \leq & 2 \cdot \ln(N+1) - \frac{(N-1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)(N+1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)}{2} \\
 & & & \text{(* Übergang zm 2er-Logarithmus *)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot \ln(2) \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1)
\end{array}
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Setzen wir diese Erkenntnis um, erhalten wir

$$
\begin{array}{rrcl}
 & \frac{T(N)}{N+1} & \leq & 2 \cdot \ln(N+1) - \frac{(N-1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)(N+1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)}{2} \\
 & & & \text{(* Übergang zm 2er-Logarithmus *)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot \ln(2) \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1)
\end{array}
$$

$$\boxed{O(N \cdot \mathrm{ld}\ N)}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N)

Setzen wir diese Erkenntnis um, erhalten wir

$$
\begin{array}{rrcl}
 & \frac{T(N)}{N+1} & \leq & 2 \cdot \ln(N+1) - \frac{(N-1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)(N+1)}{2 \cdot (N+1)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot (N+1) \cdot \ln(N+1) - \frac{(N-1)}{2} \\
 & & & \text{(* Übergang zm 2er-Logarithmus *)} \\
\Leftrightarrow & T(N) & \leq & 2 \cdot \ln(2) \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1) - \frac{(N-1)}{2} \\
\Leftrightarrow & T(N) & \leq & 1.3863 \cdot (N+1) \cdot \mathrm{ld}(N+1)
\end{array}
$$

$$T(N) \leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ \ (N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

© 2006-2023 Jörg Striegnitz

Exkurs

QuickSort: Average Case - *Herleitung der Anzahl der Vergleiche*

2. Fall: Partitionierung benötigt maximal $n+1$ Vergleiche

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

- **Zahl der Vergleiche:** bei Partitionierung mit n+1 Vergleichen
  - ***Average Case*** - Wir mitteln über alle möglichen Pivot-Elemente (sei N=2^n)

$$
\begin{aligned}
T(N) &= \frac{1}{N} \cdot \sum_{k=1}^{N} [N+1+T(k-1)+T(N-k)] \\
&= (N+1) + \frac{1}{N} \cdot \left[\sum_{k=1}^{N} T(k-1) + \sum_{k=1}^{N} T(N-k) \right] \\
&\quad \text{(* Eigentlich sind beide Summen identisch *)} \\
&= (N+1) + \frac{1}{N} \cdot [(T(0)+\cdots+T(N-1)) + (T(N-1)+\cdots+T(0))] \\
&= (N+1) + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1)
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

$$T(N) = (N+1) + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1)$$

Diese Gleichung ist aufgrund der Summe sehr unhandlich. Um die Summe zu eliminieren, wenden wir einen kleinen Trick an: Wenn $T(N)$ korrekt ist, dann muss auch die folgende Gleichung gelten (setze N = N-1)

$$T(N-1) = N + \frac{2}{N-1} \cdot \sum_{k=1}^{N-1} T(k-1)$$

Jetzt stellen wir ein **Gleichungssystem** aus beiden Gleicvhungen auf und formen dies mit dem Ziel um, eine Gleichung für $T(N)$ ohne Summenzei–chen zu erhalten:

$$\left| \begin{array}{rcl} T(N) & = & (N+1)\frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\ T(N-1) & = & N + \frac{2}{N-1} \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right|$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

Zuerst versuchen wir identische Vorfaktoren vor den Summen zu erhalten:

$$\left| \begin{array}{rcl} T(N) & = & (N+1) + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \\ T(N-1) & = & N + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right| \begin{array}{l} * \, N \\ * \, N\text{-}1 \end{array}$$

$$\Leftrightarrow \left| \begin{array}{rcl} N \cdot T(N) & = & N \cdot \left((N+1) + \frac{2}{N} \cdot \sum_{k=1}^{N} T(k-1) \right) \\ (N-1) \cdot T(N-1) & = & (N-1) \cdot \left(N + \frac{2}{(N-1)} \cdot \sum_{k=1}^{N-1} T(k-1) \right) \end{array} \right|$$

$$\Leftrightarrow \left| \begin{array}{rcl} N \cdot T(N) & = & N^2 + N + 2 \cdot \sum_{k=1}^{N} T(k-1) \\ (N-1) \cdot T(N-1) & = & N^2 - N + 2 \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right|$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

$$\left| \begin{array}{rcl} N \cdot T(N) & = & N^2 + N + 2 \cdot \sum_{k=1}^{N} T(k-1) \\ (N-1) \cdot T(N-1) & = & N^2 - N + 2 \cdot \sum_{k=1}^{N-1} T(k-1) \end{array} \right|$$

Jetzt können wir die **Summe eliminieren**, indem wir die zweite von der ersten Gleichung abziehen. Wir erhalten:

$$\begin{array}{rcl} N \cdot T(N) - (N-1) \cdot T(N-1) & = & = 2 \cdot N + 2 \cdot T(N-1) \\ N \cdot T(N) & = & 2 \cdot N + 2 \cdot T(N-1) + (N-1) \cdot T(N-1) \\ N \cdot T(N) & = & 2 \cdot N + (N+1) \cdot T(N-1) \\ T(N) & = & 2 + \dfrac{(N+1) \cdot T(N-1)}{N} \\ \dfrac{T(N)}{N+1} & = & \dfrac{2}{N+1} + \dfrac{T(N-1)}{N} \end{array}$$

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

$$\frac{T(N)}{N+1} = \frac{2}{N+1} + \frac{T(N-1)}{N}$$

In dieser Darstellung können wir leicht **sukzessive Einsetzen**:

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{2}{N+1} + \frac{T(N-1)}{N} \\
&= \frac{2}{N+1} + \frac{2}{N} + \frac{T(N-2)}{N-1} \\
&= \frac{2}{N+1} + \frac{2}{N} + \frac{2}{N-1} + \frac{T(N-3)}{N-2} \\
&= \vdots \\
&= 2 \cdot \sum_{k=3}^{N+1} \frac{1}{k} + \frac{T(1)}{2} \\
&= 2 \cdot \sum_{k=3}^{N+1} \frac{1}{k}
\end{aligned}$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N+1)

Mit unserer Abschätzung für die **Harmonische Reihe** (siehe Folie 253) gelangen wir zu:

$$\begin{aligned} \frac{T(N)}{N+1} &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} \\ &\leq 2 \cdot \ln(N+1) \end{aligned}$$

und weiter:

$$\begin{aligned} T(N) &\leq 2 \cdot \ln 2 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \\ &\leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \end{aligned}$$

Folglich macht es unter reinen Komplexitätsbetrachtung keinen Unterschied, ob wir n oder n+1 mal beim Partitionieren vergleichen!

© 2006-2023 Jörg Striegnitz

QuickSort: Analyse des Average Case (N+1)

Mit unserer Abschätzung für die **Harmonische Reihe** (siehe Folie 253) gelangen wir zu:

$$\begin{aligned} \frac{T(N)}{N+1} &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} \\ &\leq 2 \cdot \ln(N+1) \end{aligned}$$

und weiter:

$$\begin{aligned} T(N) &\leq 2 \cdot \ln 2 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \\ &\leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \end{aligned}$$

Folglich macht es unter reinen Komplexitätsbetrachtung keinen Unterschied, ob wir n oder n+1 mal beim Partitionieren vergleichen!

$$\boxed{O(N \cdot \mathrm{ld}\ N)}$$

© 2006-2023 Jörg Striegnitz

`QuickSort`: Analyse des Average Case (N+1)

Mit unserer Abschätzung für die **Harmonische Reihe** (siehe Folie 253) gelangen wir zu:

$$\begin{aligned} \frac{T(N)}{N+1} &= 2 \cdot \sum_{i=3}^{N+1} \frac{1}{i} \\ &\leq 2 \cdot \ln(N+1) \end{aligned}$$

und weiter:

$$\begin{aligned} T(N) &\leq 2 \cdot \ln 2 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \\ &\leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ (N+1) \end{aligned}$$

Folglich macht es unter reinen Komplexitätsbetrachtung keinen Unterschied, ob wir n oder n+1 mal beim Partitionieren vergleichen!

$$T(N) \leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ (N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

© 2006-2023 Jörg Striegnitz

Komplexität von `Quicksort`: Zusammenfassung

- **Zusammenfassend ergibt sich für die Zahl der Vergleiche**

© 2006-2023 Jörg Striegnitz

Komplexität von `Quicksort`: Zusammenfassung

- **Zusammenfassend ergibt sich für die Zahl der Vergleiche**
  - ***Best Case***

$$T(N) = (N+1) \cdot \mathrm{ld}\,(N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

© 2006-2023 Jörg Striegnitz

Komplexität von `Quicksort`: Zusammenfassung

- **Zusammenfassend ergibt sich für die Zahl der Vergleiche**
  - ***Best Case***

$$T(N) = (N+1) \cdot \mathrm{ld}\,(N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

  - ***Average Case***

$$T(N) \leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ \ (N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

© 2006-2023 Jörg Striegnitz

Komplexität von `Quicksort`: Zusammenfassung

- **Zusammenfassend ergibt sich für die Zahl der Vergleiche**
  - ***Best Case***

$$T(N) = (N+1) \cdot \mathrm{ld}\,(N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

  - ***Average Case***

$$T(N) \leq 1.386 \cdot (N+1) \cdot \mathrm{ld}\ \ (N+1)$$

$$O(N \cdot \mathrm{ld}\ N)$$

  - ***Worst Case***

$$T(N) \leq \frac{(N+1) \cdot (N+2)}{2} - 3$$

$$O(N^2)$$

© 2006-2023 Jörg Striegnitz

Varianten und Verbesserungen

© 2006-2023 Jörg Striegnitz

Varianten und Verbesserungen

- **Geschicktere Wahl des Pivot-Elements**
  - pivot=$(a[l]+a[r])/2$
    Vorsicht! Dann Pivot u.U. nicht mehr in F - Implementierung anpassen!
  - ***Median of Three*** - Wähle mittleres von drei zufällig ausgewählten Elementen
    Systematisch ungünstige Zerlegungen werden unwahrscheinlich - *worst case* wird unwahrscheinlicher - **Vorsicht!** Pivot ggf. nicht mehr rechtes Sentinel!

© 2006-2023 Jörg Striegnitz

Varianten und Verbesserungen

- **Geschicktere Wahl des Pivot-Elements**
  - pivot=$(a[l]+a[r])/2$
    Vorsicht! Dann Pivot u.U. nicht mehr in F - Implementierung anpassen!
  - ***Median of Three*** - Wähle mittleres von drei zufällig ausgewählten Elementen
    Systematisch ungünstige Zerlegungen werden unwahrscheinlich - *worst case* wird unwahrscheinlicher - **Vorsicht!** Pivot ggf. nicht mehr rechtes Sentinel!
- **Sonderbehandlung kleiner Listen**
  - Für kleine Felder ist `QuickSort` wegen der rekursiven Struktur nicht effizient
  - **Abhilfe**: Zu Beginn Testen, ob `(l-r)` größer als eine Schranke `M` ist. Wenn ja, `QuickSort`, sonst z.B. `Insertion Sort` verwenden.

© 2006-2023 Jörg Striegnitz

Varianten und Verbesserungen

- **Geschicktere Wahl des Pivot-Elements**
  - pivot=$(a[l]+a[r])/2$
    Vorsicht! Dann Pivot u.U. nicht mehr in F - Implementierung anpassen!
  - ***Median of Three*** - Wähle mittleres von drei zufällig ausgewählten Elementen
    Systematisch ungünstige Zerlegungen werden unwahrscheinlich - *worst case* wird unwahrscheinlicher - **Vorsicht!** Pivot ggf. nicht mehr rechtes Sentinel!
- **Sonderbehandlung kleiner Listen**
  - Für kleine Felder ist `QuickSort` wegen der rekursiven Struktur nicht effizient
  - **Abhilfe**: Zu Beginn Testen, ob (`l-r`) größer als eine Schranke `M` ist. Wenn ja, `QuickSort`, sonst z.B. `Insertion Sort` verwenden.
- **Reduktion des Speicherbedarfs**
  - Der Speicherbedarf wird indirekt über die Größe der Aktivierungsblöcke bestimmt. Wenn stets zuerst das kleinere der Teilarrays bearbeitet wird, ist die Größe der Aktivierungsblöcke nach oben durch $2 \cdot \text{ld}\, N$ beschränkt.

© 2006-2023 Jörg Striegnitz

Varianten und Verbesserungen

- **Geschicktere Wahl des Pivot-Elements**
  - pivot=$(a[l]+a[r])/2$
    Vorsicht! Dann Pivot u.U. nicht mehr in F - Implementierung anpassen!
  - ***Median of Three*** - Wähle mittleres von drei zufällig ausgewählten Elementen
    Systematisch ungünstige Zerlegungen werden unwahrscheinlich - *worst case* wird unwahrscheinlicher - **Vorsicht!** Pivot ggf. nicht mehr rechtes Sentinel!
- **Sonderbehandlung kleiner Listen**
  - Für kleine Felder ist `QuickSort` wegen der rekursiven Struktur nicht effizient
  - **Abhilfe**: Zu Beginn Testen, ob `(l-r)` größer als eine Schranke `M` ist. Wenn ja, `QuickSort`, sonst z.B. `Insertion Sort` verwenden.
- **Reduktion des Speicherbedarfs**
  - Der Speicherbedarf wird indirekt über die Größe der Aktivierungsblöcke bestimmt. Wenn stets zuerst das kleinere der Teilarrays bearbeitet wird, ist die Größe der Aktivierungsblöcke nach oben durch $2 \cdot \text{ld}\, N$ beschränkt.
- **Iterative Variante**
  - Ist möglich, erfordert aber einen Stack

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

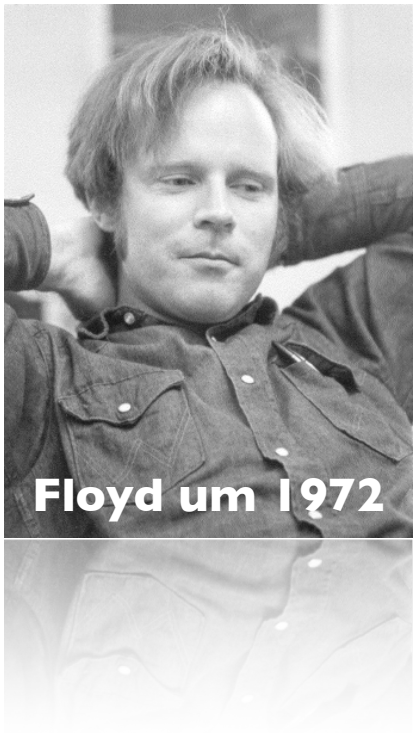
4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- **4.2.2. Heapsort**
- 4.2.3. Merge Sort

4.3.Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

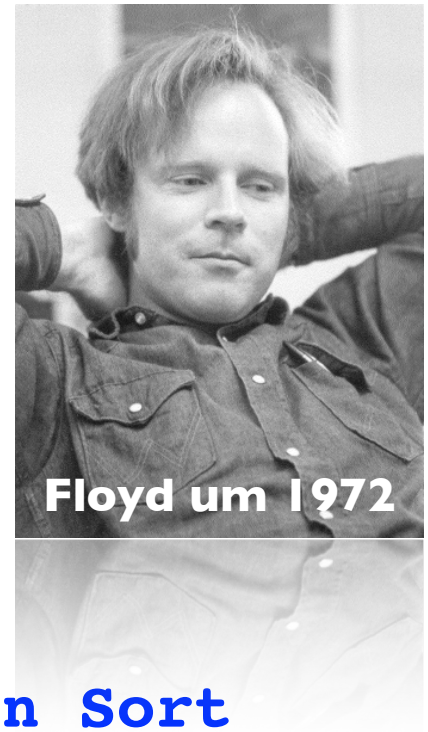
Heapsort

- **1964 veröffentlicht von J.W.J. Williams verbessert von Robert W. Floyd**



© 2006-2023 Jörg Striegnitz

Heapsort

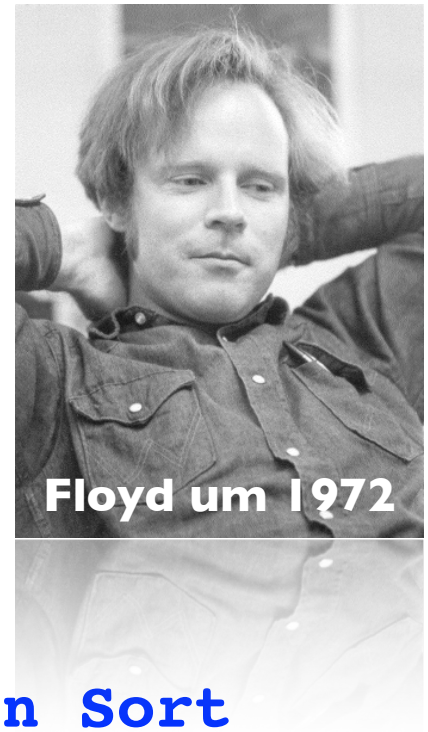


- **1964 veröffentlicht von J.W.J. Williams verbessert von Robert W. Floyd**

- **Es handelt sich um die Erweiterung von `Selection Sort` mittels eines Heap** (dadurch schnelleres Auffinden des Maximum / Minimum möglich)

© 2006-2023 Jörg Striegnitz

Heapsort



- **1964 veröffentlicht von J.W.J. Williams verbessert von Robert W. Floyd**

- **Es handelt sich um die Erweiterung von `Selection Sort` mittels eines Heap** (dadurch schnelleres Auffinden des Maximum / Minimum möglich)

- **Zentral ist der Begriff der Heap-Eigenschaft**

© 2006-2023 Jörg Striegnitz

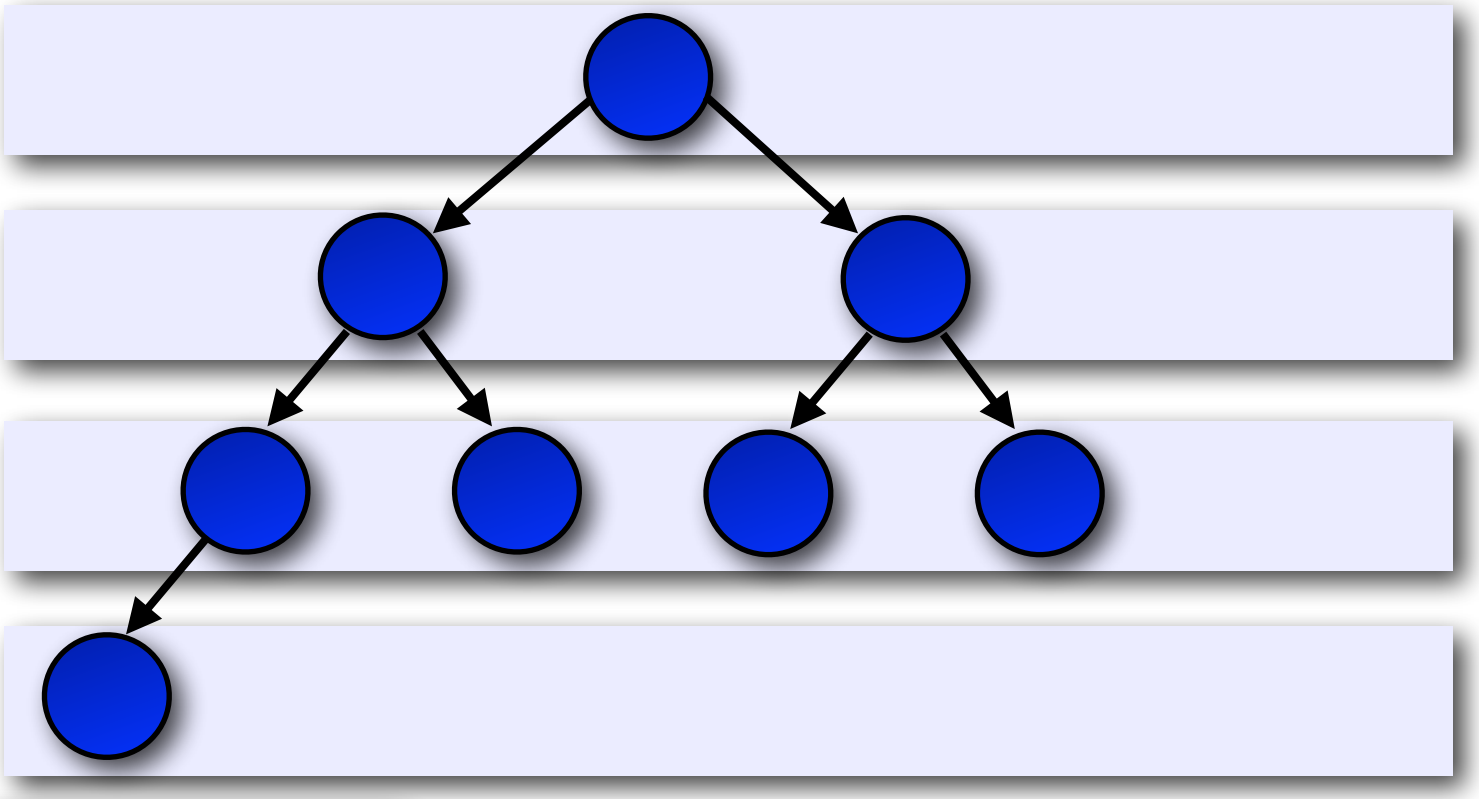
Was ist ein Heap?

- Ein **Heap** ist ein **links-vollständiger Binärbaum**

© 2006-2023 Jörg Striegnitz

Was ist ein Heap?

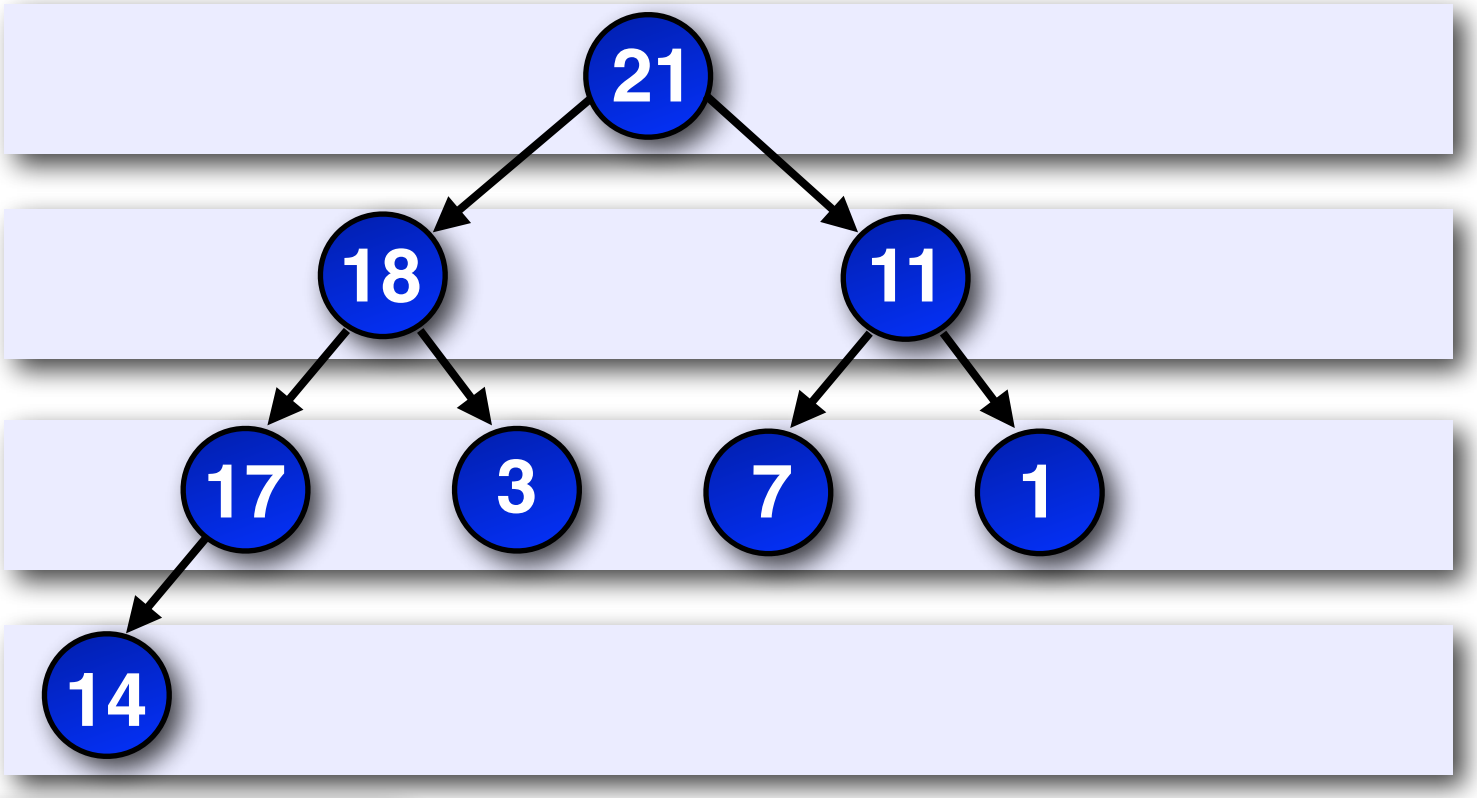
- Ein **Heap** ist ein **links-vollständiger Binärbaum**



© 2006-2023 Jörg Striegnitz

Was ist ein Heap?

- Ein **Heap** ist ein **links-vollständiger Binärbaum**
- Knoten enthalten die Schlüssel des zu sortierenden Feldes

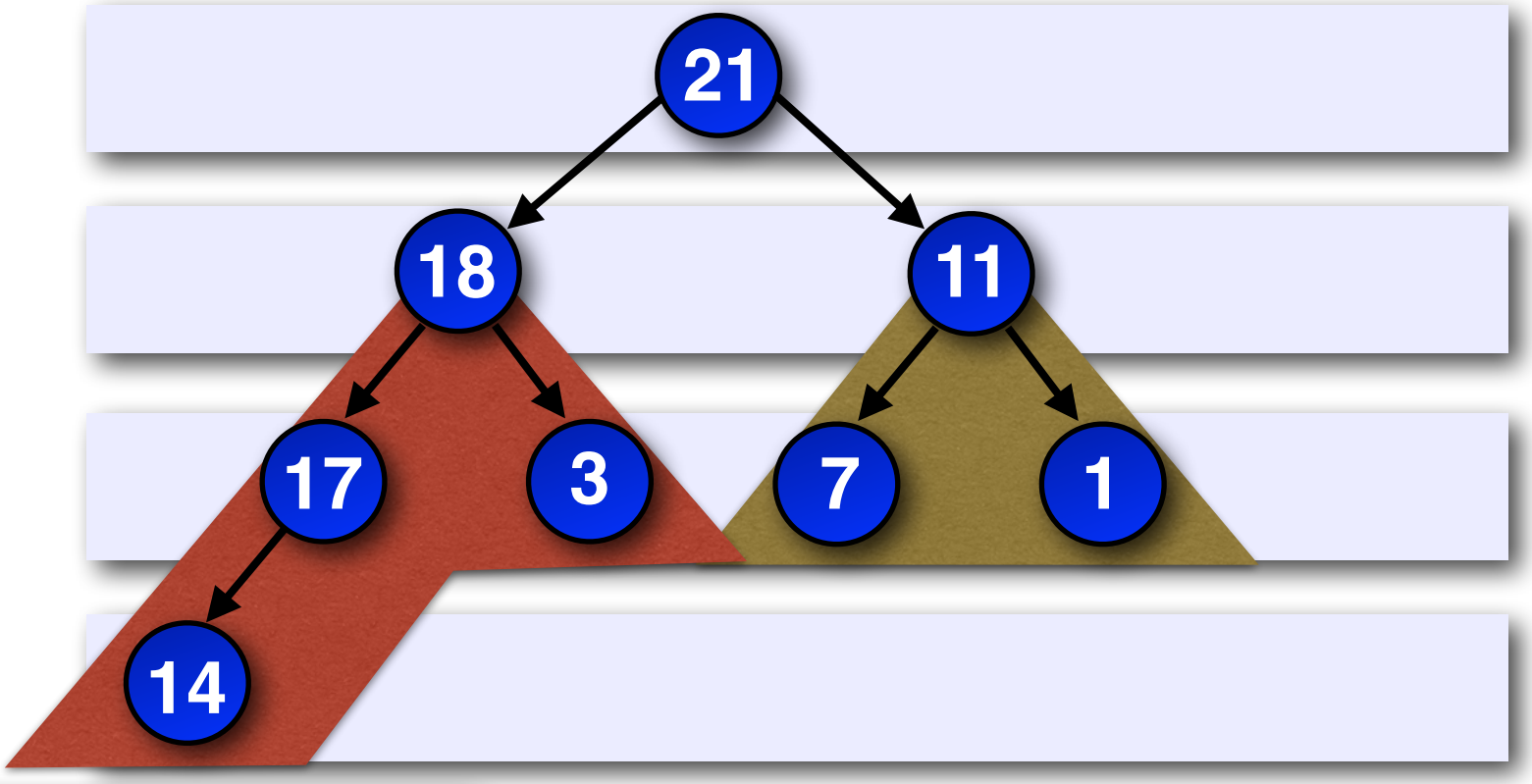


© 2006-2023 Jörg Striegnitz

Was ist ein Heap?

- Ein **Heap** ist ein **links-vollständiger Binärbaum**
- Knoten enthalten die Schlüssel des zu sortierenden Feldes
- Baum erfüllt **Struktur-Invariante:**

Für alle Teilbäume gilt:
an der Wurzel steht der **maximale** Schlüsselwert



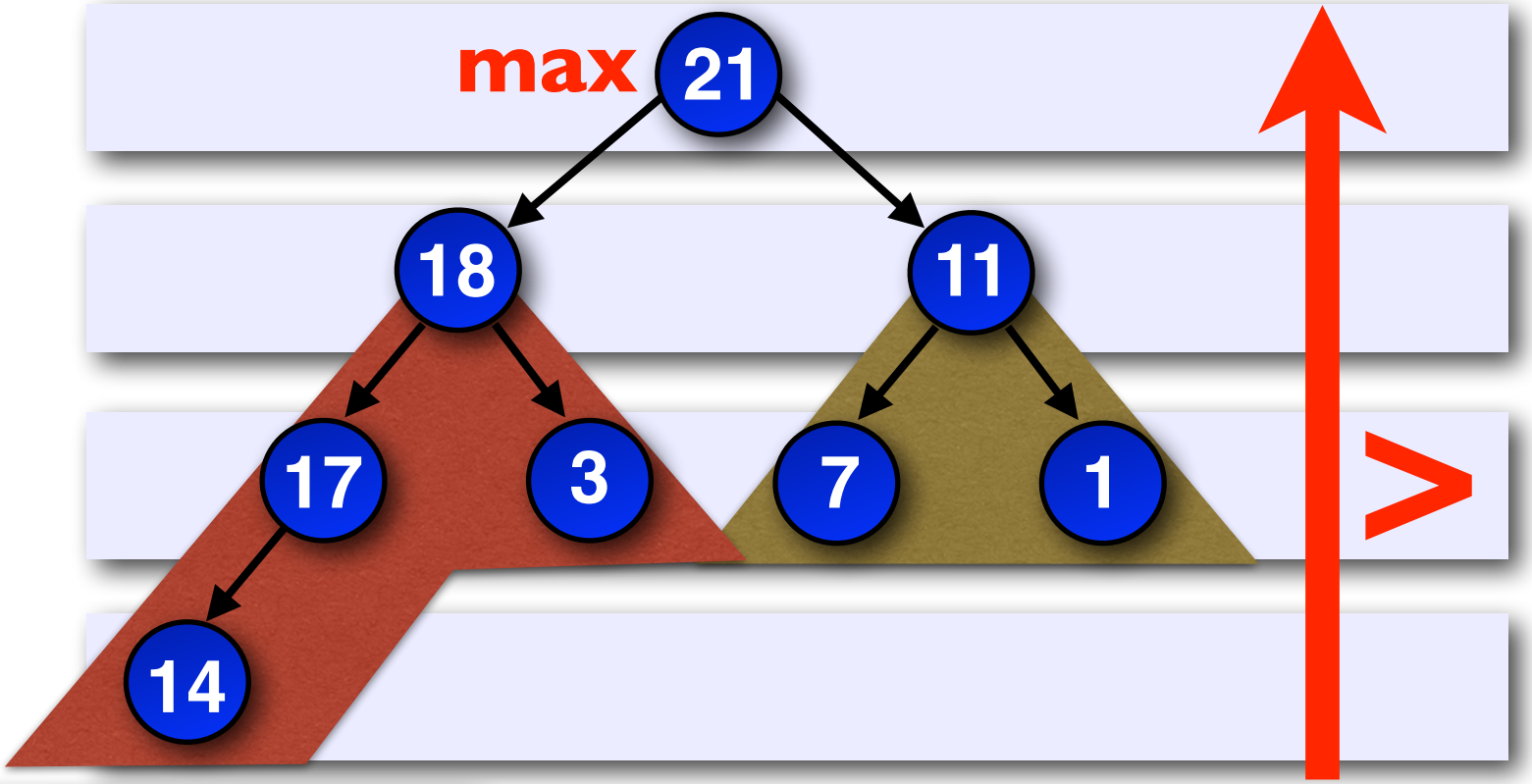
© 2006-2023 Jörg Striegnitz

Was ist ein Heap?

- Ein **Heap** ist ein **links-vollständiger Binärbaum**
- Knoten enthalten die Schlüssel des zu sortierenden Feldes
- Baum erfüllt **Struktur-Invariante:**

Für alle Teilbäume gilt:
an der Wurzel steht der **maximale** Schlüsselwert

- **Folge**: Maximaler Schlüsselwert an Wurzel des Baumes!



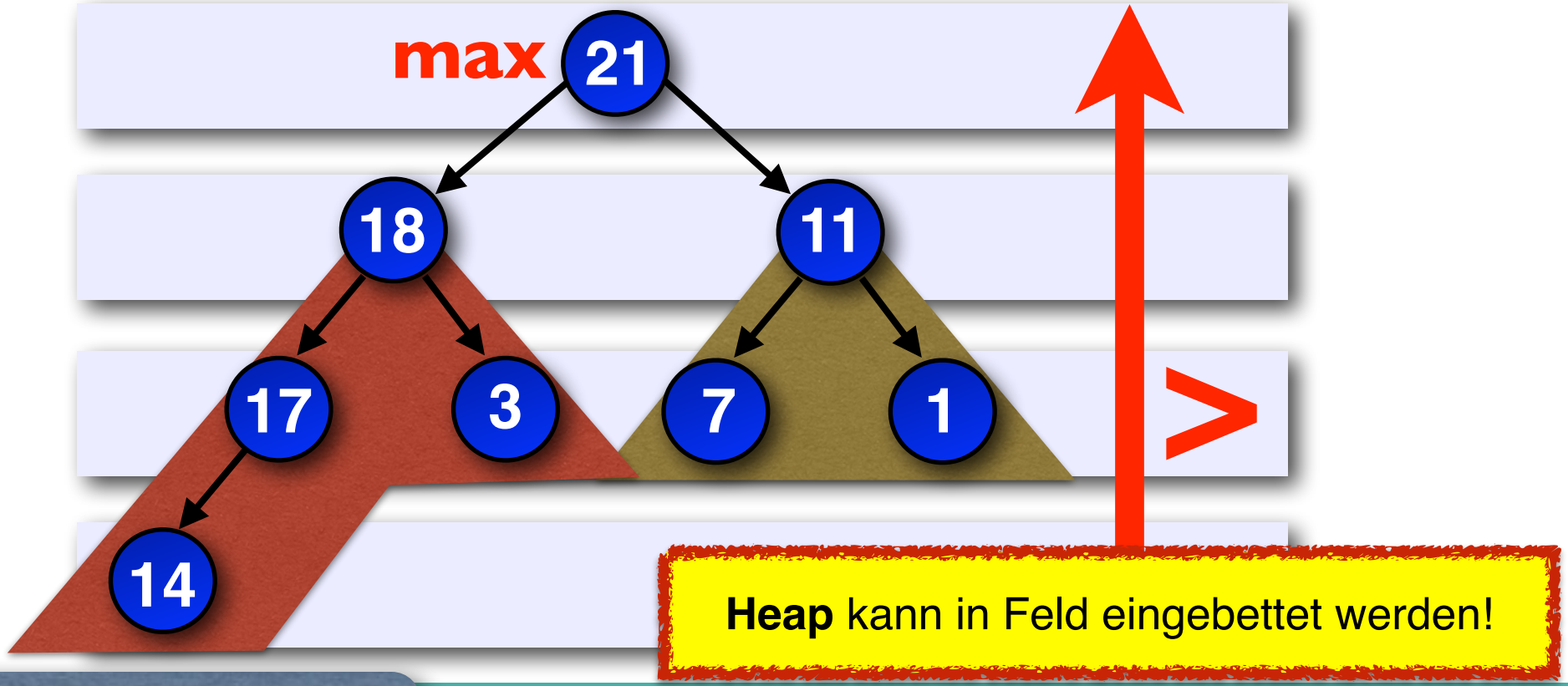
© 2006-2023 Jörg Striegnitz

Was ist ein Heap?

- Ein **Heap** ist ein **links-vollständiger Binärbaum**
- Knoten enthalten die Schlüssel des zu sortierenden Feldes
- Baum erfüllt **Struktur-Invariante:**

Für alle Teilbäume gilt:
an der Wurzel steht der **maximale** Schlüsselwert

- **Folge**: Maximaler Schlüsselwert an Wurzel des Baumes!



© 2006-2023 Jörg Striegnitz

Heap und Heap-Eigenschaft

D Heap / Heap-Eigenschaft

Ein **Heap** ist ein links-vollständiger Binärbaum, der in ein Feld eingebettet ist:

- Ein Feld $a[0..N-1]$ erfüllt die **Heap-Eigenschaft**, wenn gilt:

$$a\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right] \geq a[i] \qquad \text{für } i = 1, \ldots, N-1$$

- Ein Feld ist ein **Heap beginnend in Position** $l = 0, \ldots, N-1$, falls gilt:

$$a\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right] \geq a[i] \qquad \text{für } i = 2l+1, \ldots, N-1$$

© 2006-2023 Jörg Striegnitz

Binärbaum: Array-Einbettung (Wiederholung)

D Vollständiger Binärbaum

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist (auch links-vollständig).

Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

Vorgänger: $\mathrm{Pred}(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

Nachfolger: $\mathrm{succ(n)} = \begin{cases} 2n+1 & \text{linker Sohn} \\ 2n+2 & \text{rechter Sohn} \end{cases}$

© 2006-2023 Jörg Striegnitz

Binärbaum: Array-Einbettung (Wiederholung)

D **Vollständiger Binärbaum**

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist (auch links-vollständig).

Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

Vorgänger: $\mathrm{Pred}(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

Nachfolger: $\mathrm{succ(n)} = \begin{cases} 2n+1 & \text{linker Sohn} \\ 2n+2 & \text{rechter Sohn} \end{cases}$

Die Heap-Eigenschaft besagt, dass der Schlüssel jedes Knotens größer ist als der seiner Söhne; insbesondere gilt:

© 2006-2023 Jörg Striegnitz

Binärbaum: Array-Einbettung (Wiederholung)

D Vollständiger Binärbaum

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist (auch links-vollständig).

Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

Vorgänger: $\mathrm{Pred}(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

Nachfolger: $\mathrm{succ(n)} = \begin{cases} 2n+1 & \text{linker Sohn} \\ 2n+2 & \text{rechter Sohn} \end{cases}$

Die Heap-Eigenschaft besagt, dass der Schlüssel jedes Knotens größer ist als der seiner Söhne; insbesondere gilt:

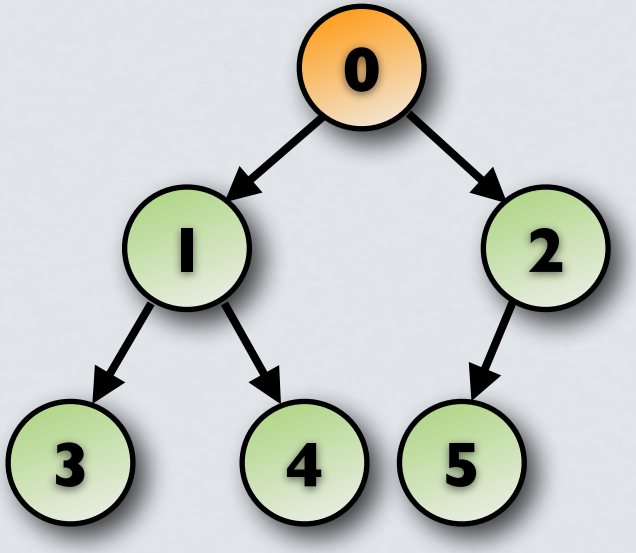
$$a[0] = \max\{\, a[i] \mid i=0,\ldots,N\text{-}1 \,\}$$

© 2006-2023 Jörg Striegnitz

Binärbaum: Array-Einbettung (Wiederholung)

D Vollständiger Binärbaum

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist (auch links-vollständig).



Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

Vorgänger: $\mathrm{Pred}(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

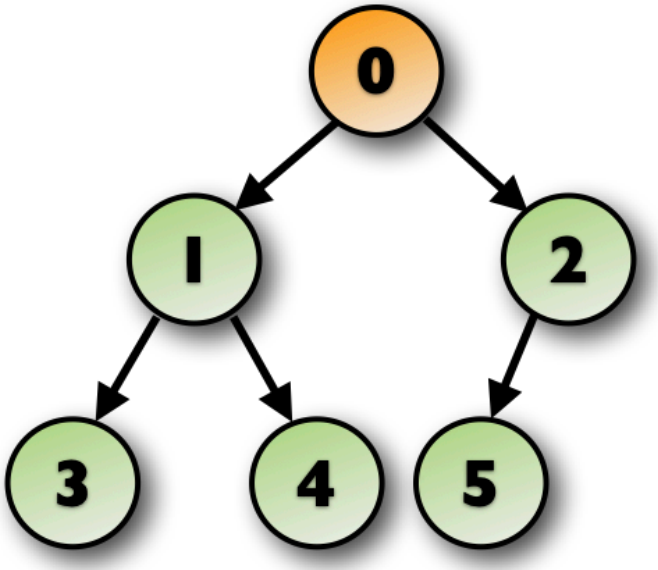
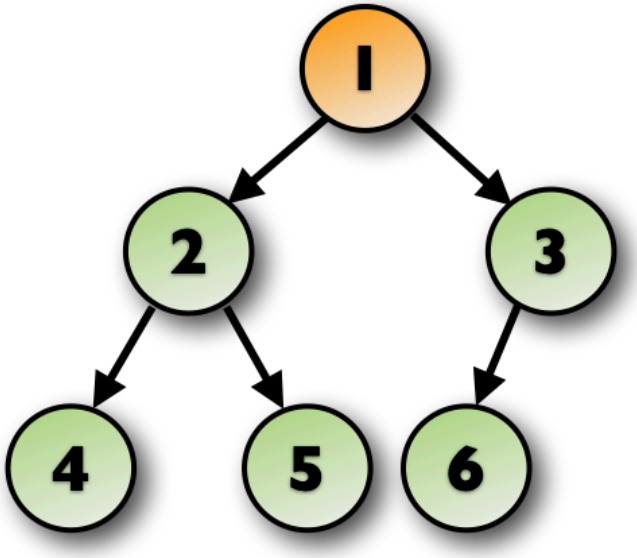
Nachfolger: $\mathrm{succ(n)} = \begin{cases} 2n+1 & \text{linker Sohn} \\ 2n+2 & \text{rechter Sohn} \end{cases}$

Die Heap-Eigenschaft besagt, dass der Schlüssel jedes Knotens größer ist als der seiner Söhne; insbesondere gilt:

$$a[0] = \max\{\, a[i] \mid i=0,\ldots,N\text{-}1 \,\}$$

© 2006-2023 Jörg Striegnitz

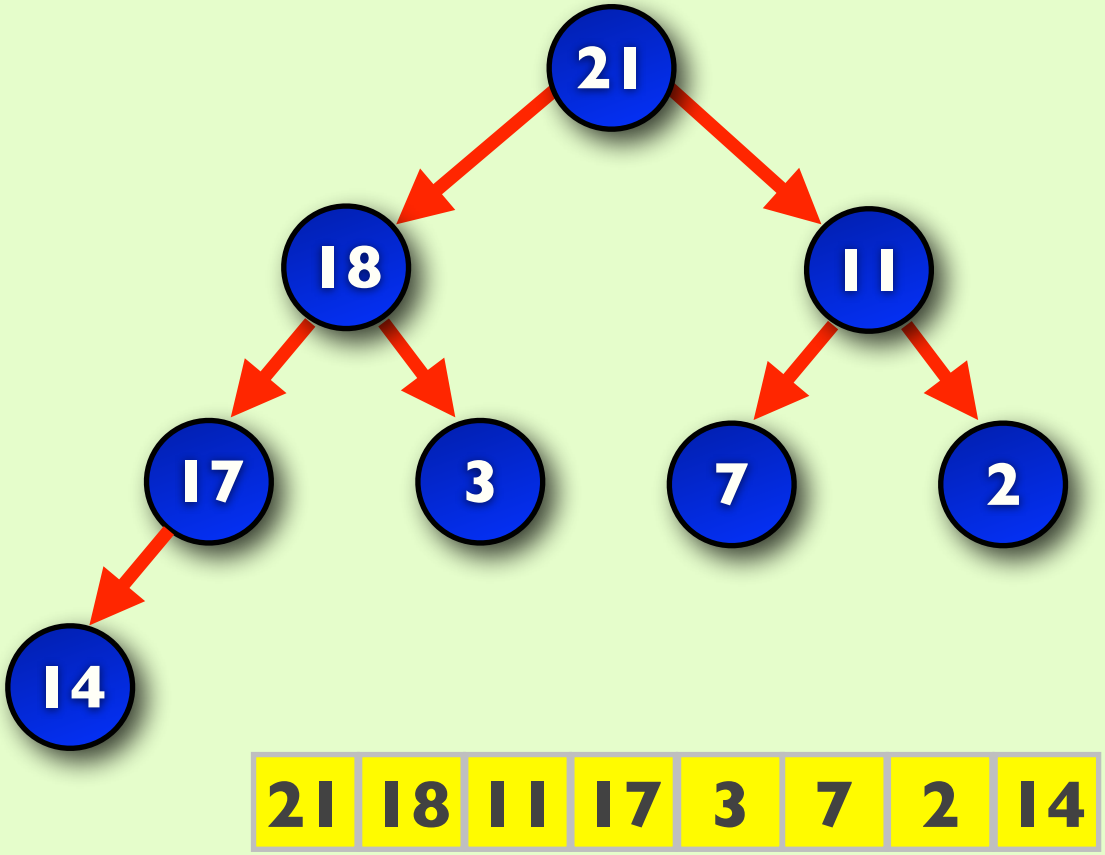
Heap / Einbettung Binärbaum in Feld

	`a[0..N-1]`	`a[1..N]`
Vorgänger	$\left\lfloor \frac{n-1}{2} \right\rfloor$	$\left\lfloor \frac{n}{2} \right\rfloor$
Linker Sohn	$2 \cdot n + 1$	$2 \cdot n$
Rechter Sohn	$2 \cdot n + 2$	$2 \cdot n + 1$
Heap-Eigenschaft ab Index	$\left\lfloor \frac{(\mathtt{N}-1)+1}{2} \right\rfloor$	$\left\lceil \frac{\mathtt{N}+1}{2} \right\rceil$
Beispiel		

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

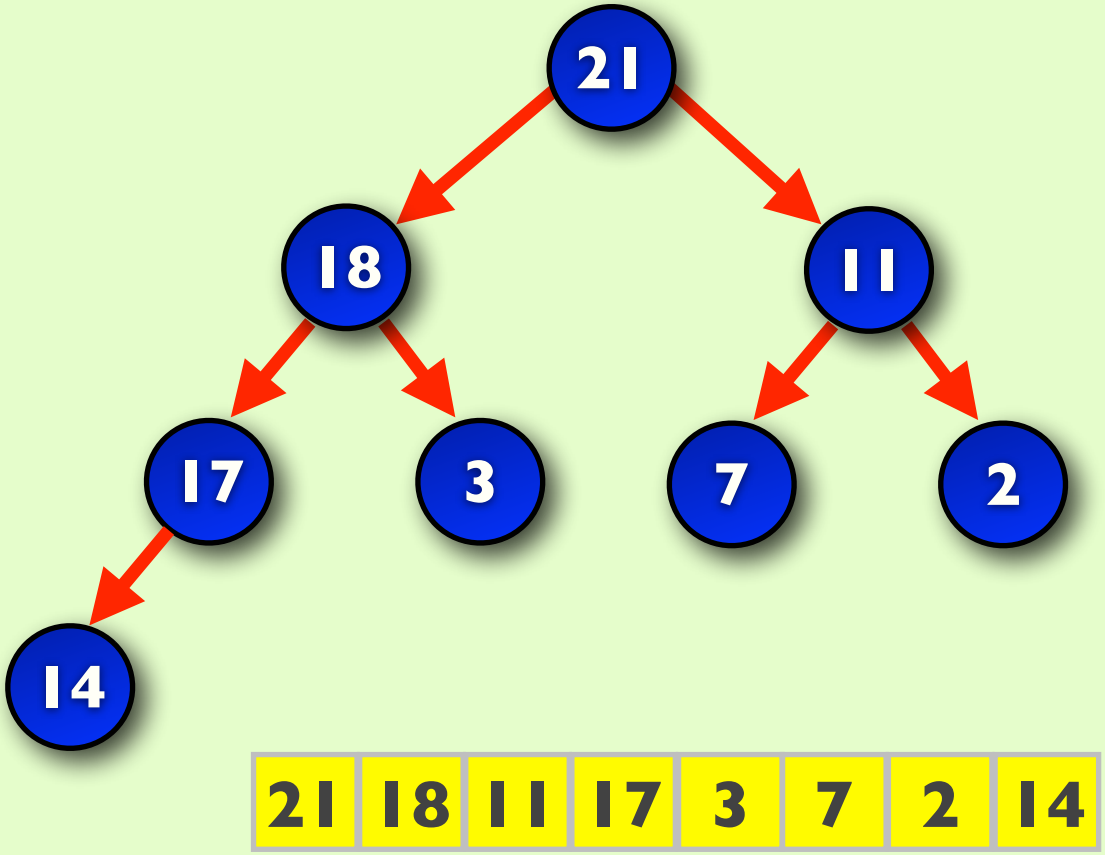
- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**



© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**

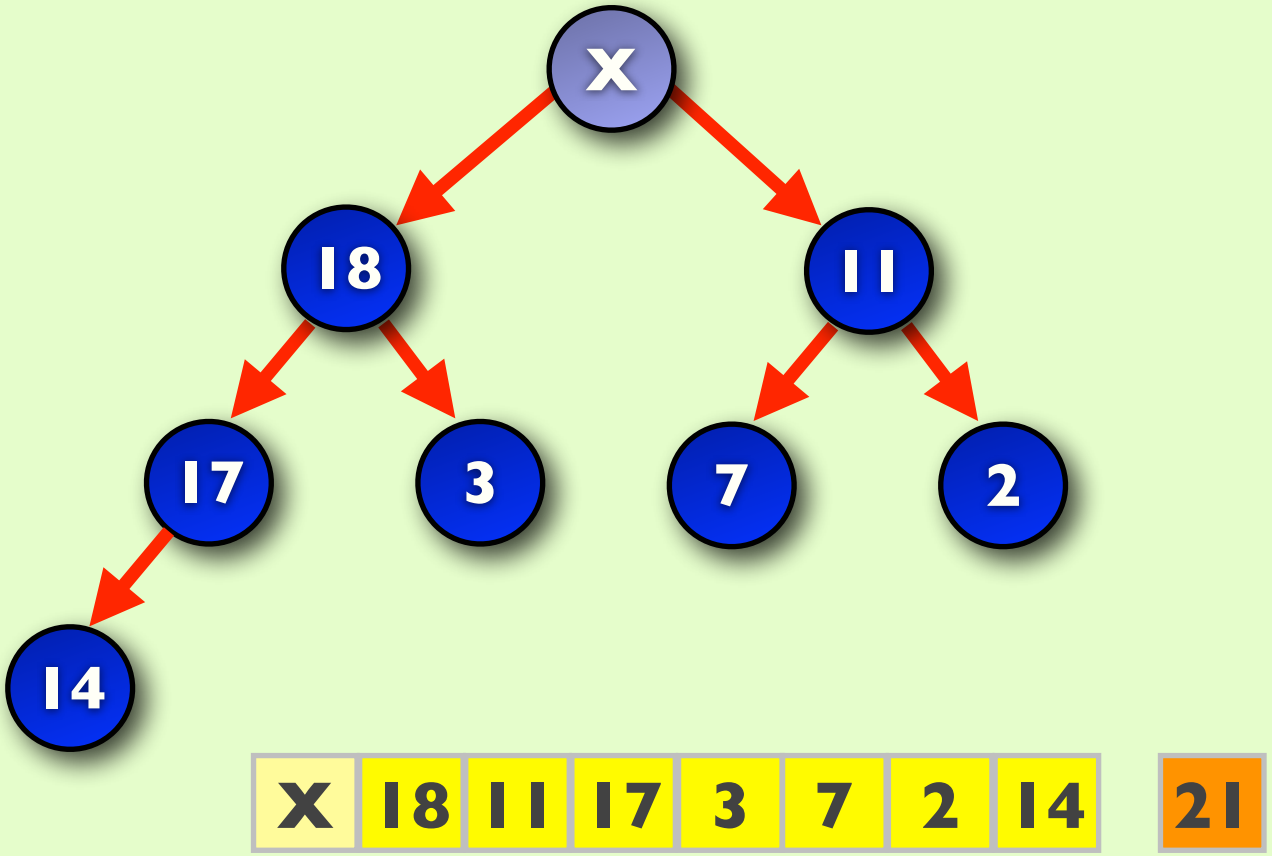


1 Wurzel löschen und in Hilfsvariable merken

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**

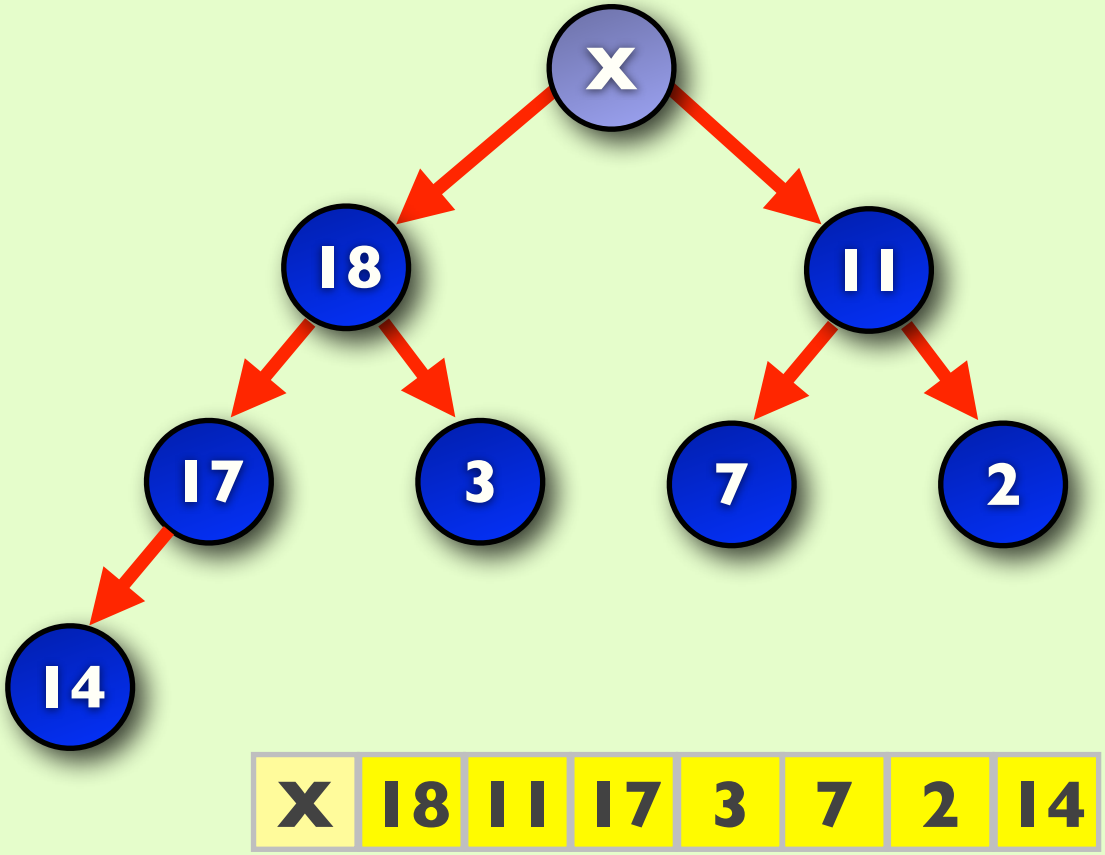


1 Wurzel löschen und in Hilfsvariable merken

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**

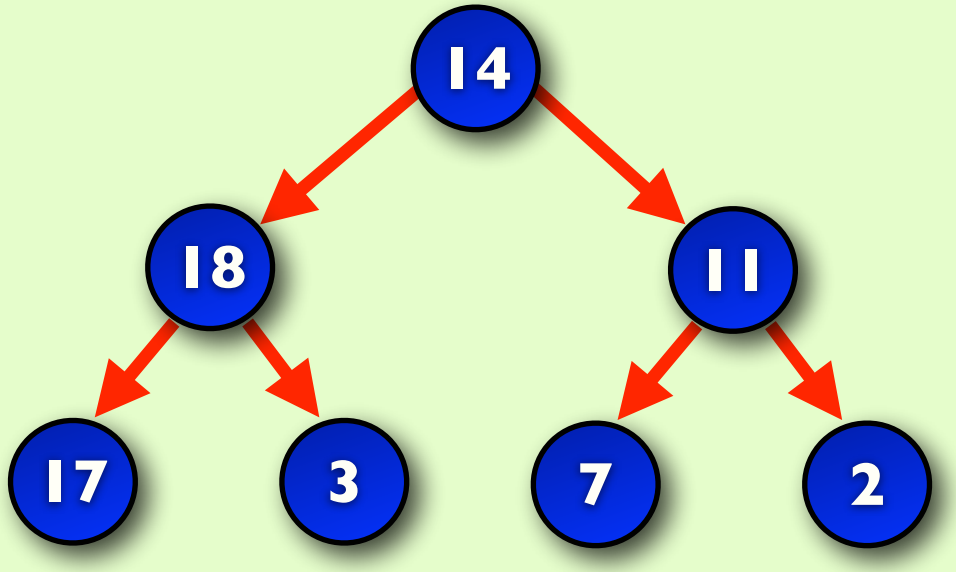


2 Kopiere letztes Element des Feldes an Position der Wurzel

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**



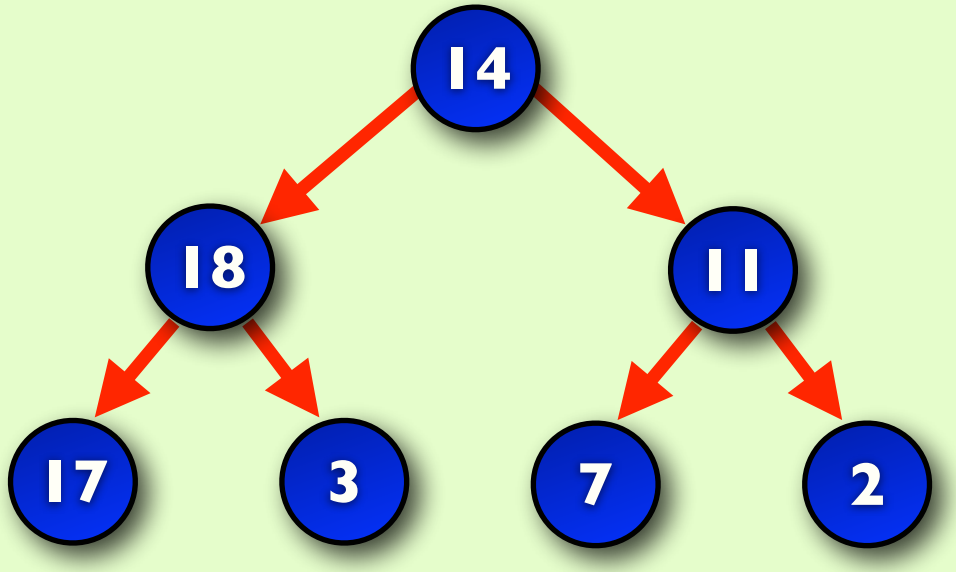
14	18	11	17	3	7	2	X	21

2 Kopiere letztes Element des Feldes an Position der Wurzel

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**



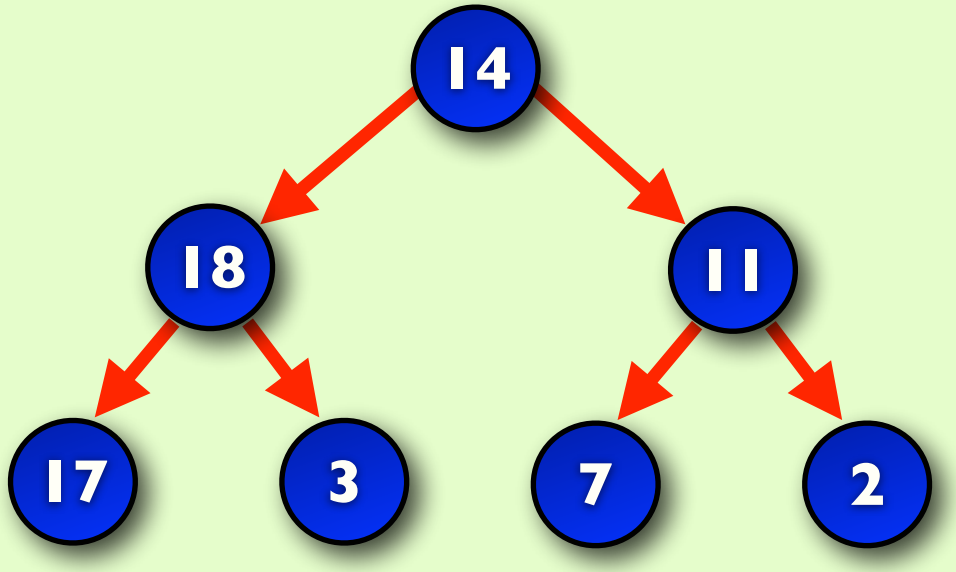
14	18	11	17	3	7	2	X	21

2 Kopiere letztes Element des Feldes an Position der Wurzel - **so wird Platz für das größte Element!**

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**



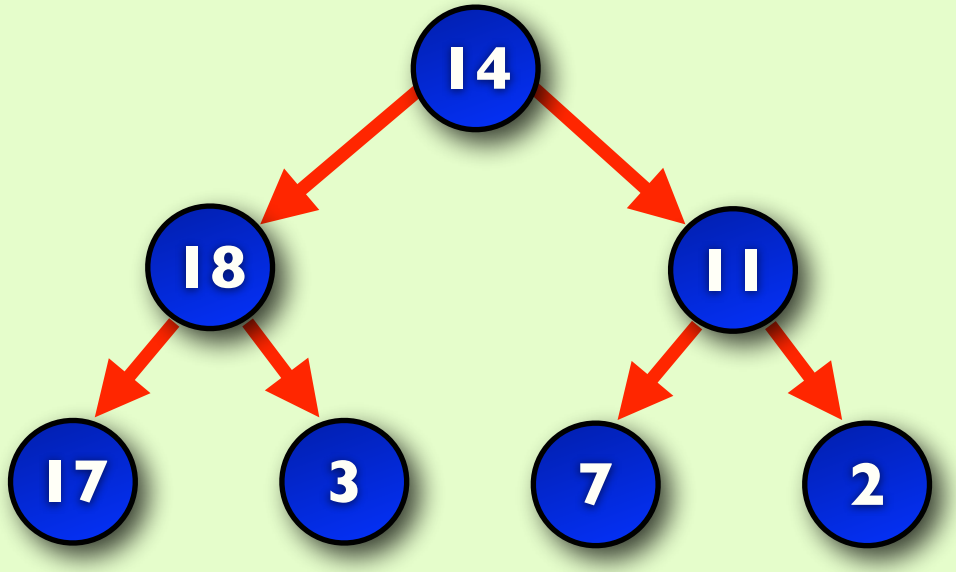
14	18	11	17	3	7	2	21

2 Kopiere letztes Element des Feldes an Position der Wurzel - **so wird Platz für das größte Element!**

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **kopieren das Maximum an das Ende des Feldes**



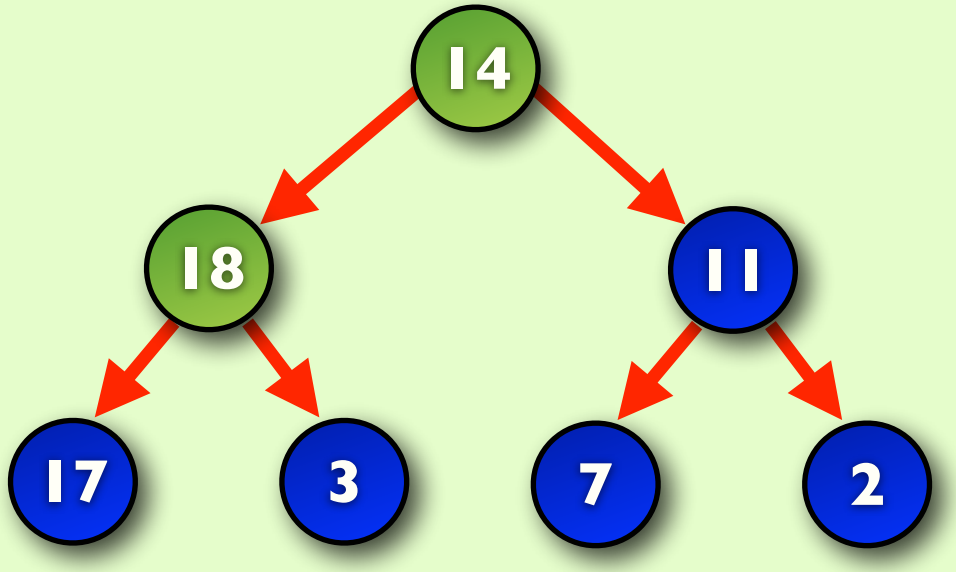
14	18	11	17	3	7	2	21

3 Vertausche **14** solange mit dem **größten Sohn**, bis alle Sohnknoten nur noch kleinere Elemente enthalten

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **und kopieren das Maximum an das Ende des Feldes**



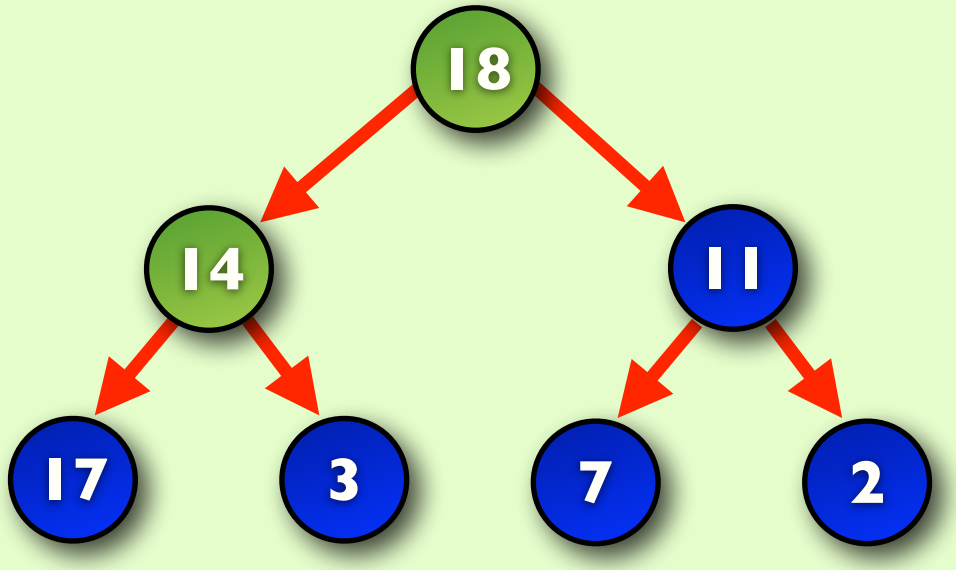
14	18	11	17	3	7	2	21

3 Vertausche **14** solange mit dem **größten Sohn**, bis alle Sohnknoten nur noch kleinere Elemente enthalten

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **und kopieren das Maximum an das Ende des Feldes**



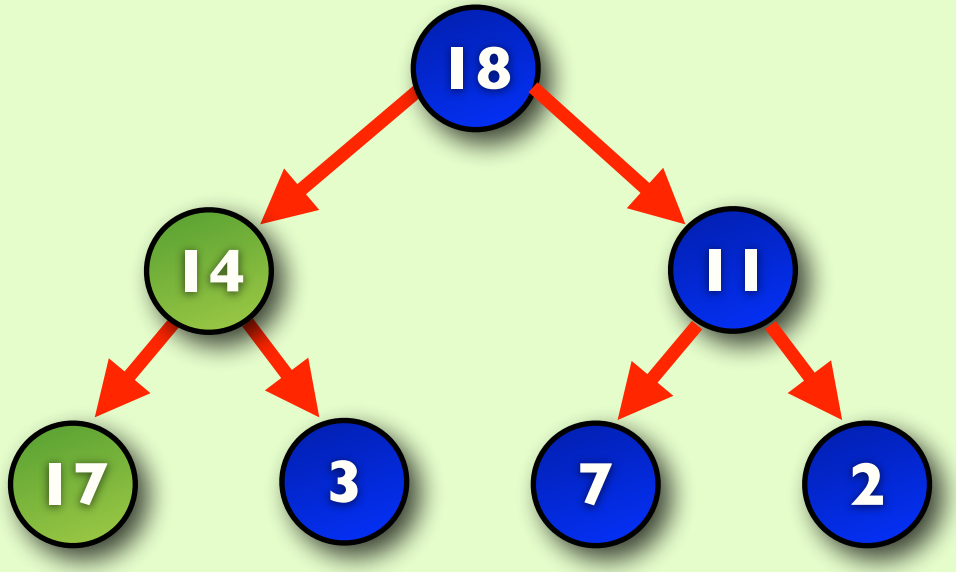
14	18	11	17	3	7	2	21

3 Vertausche **14** solange mit dem **größten Sohn**, bis alle Sohnknoten nur noch kleinere Elemente enthalten

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **und kopieren das Maximum an das Ende des Feldes**



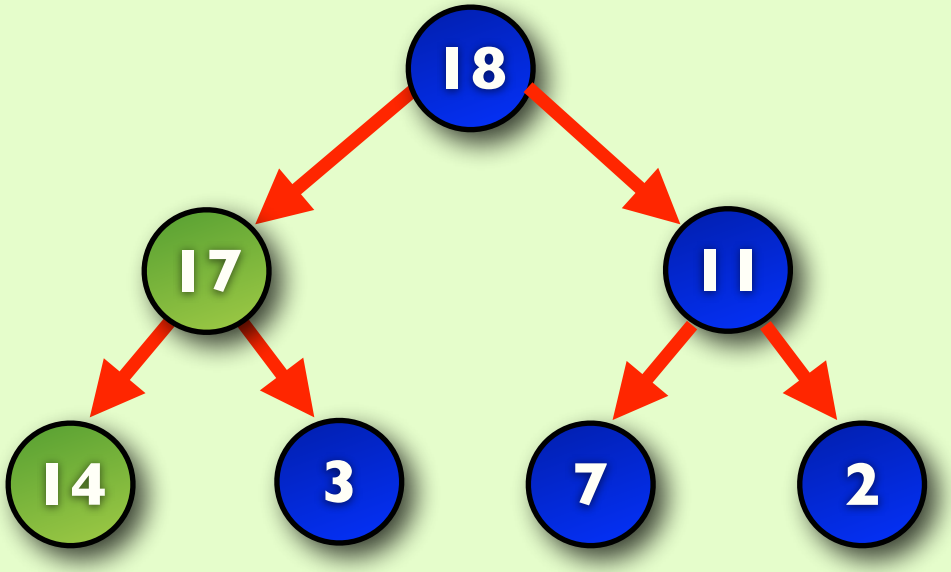
18	14	11	17	3	7	2	21

3 Vertausche **14** solange mit dem **größten Sohn**, bis alle Sohnknoten nur noch kleinere Elemente enthalten

© 2006-2023 Jörg Striegnitz

Die Operation *ReHeap*

- **Wir löschen die Wurzel,**
- **stellen die Heap-Eigenschaft wieder her und**
- **und kopieren das Maximum an das Ende des Feldes**



18	17	11	14	3	7	2	21

3 Vertausche **14** solange mit dem **größten Sohn**, bis alle Sohnknoten nur noch kleinere Elemente enthalten

© 2006-2023 Jörg Striegnitz

`HeapSort`: Der Algorithmus

- **Das Prinzip:**
  - **1. Schritt Heap-Aufbau**
    Wandle das Feld `a[0..N-1]` in einen Heap um
  - **2. Schritt Heap-Select-Max**:
    `For I:=0 to N-2 do`
    - Tausche `a[0]` (die Wurzel) und `a[N-1-i]`
    - Stelle für das Restfeld `a[0..N-2-i]` die Heap-Eigenschaft wieder her

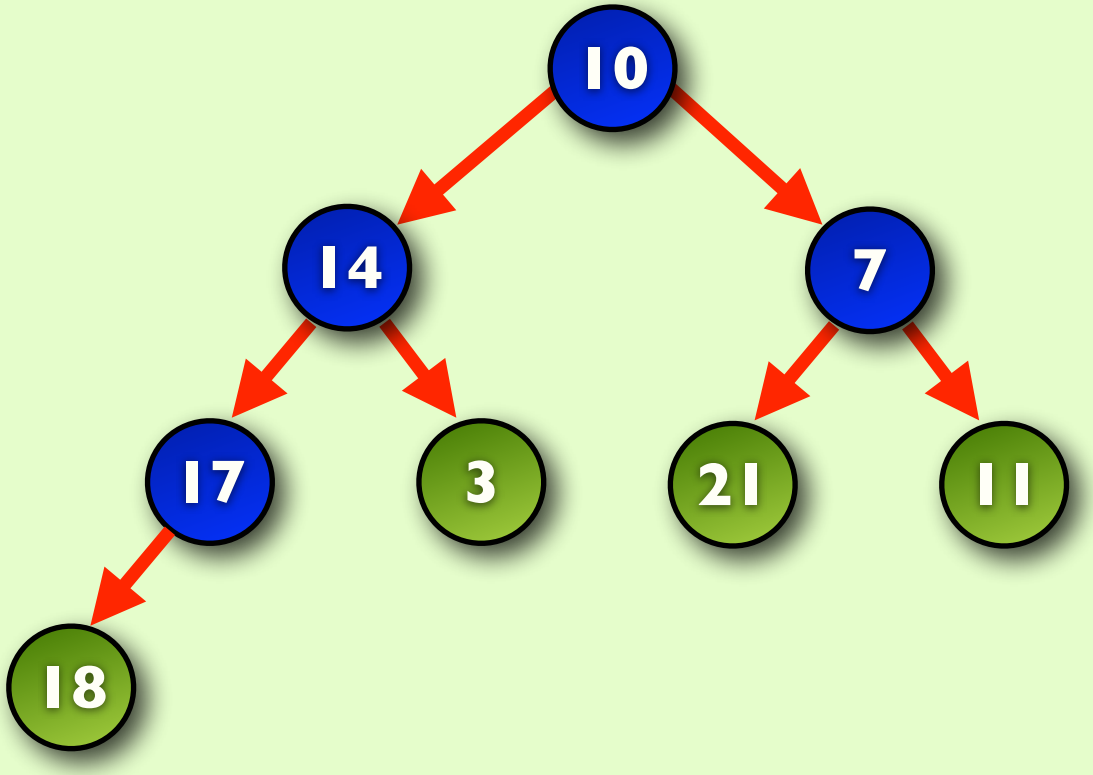
- **Aufbau eines Heap:**
  - Ab Position $l = \lfloor (N+1)/2 \rfloor$ erfüllt jedes Feld die Heap-Eigenschaft

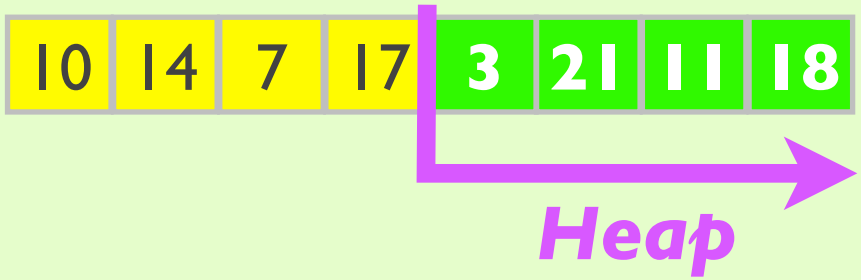
    Grund: ab dieser Position haben Knoten keine Kinder (s.n.F.)!

© 2006-2023 Jörg Striegnitz

B Aufbau eines Heap

Ab der Position $\lfloor (N+1)/2 \rfloor$ ist jedes Teilbaum ein Heap:



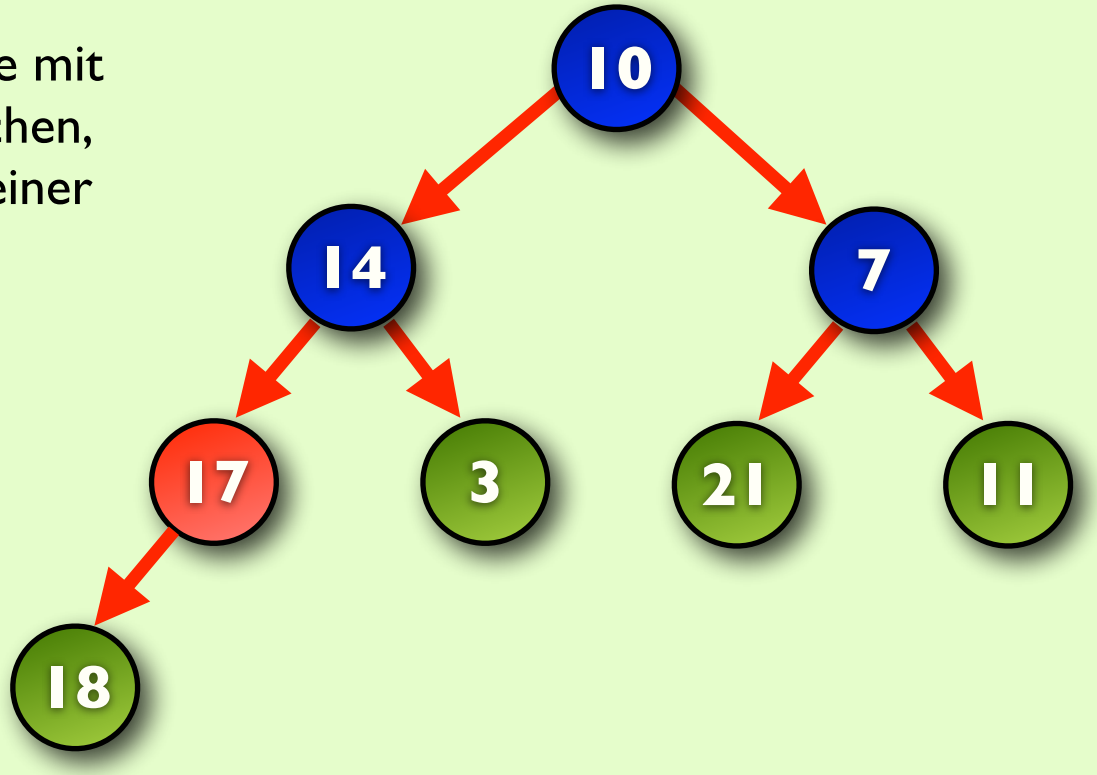


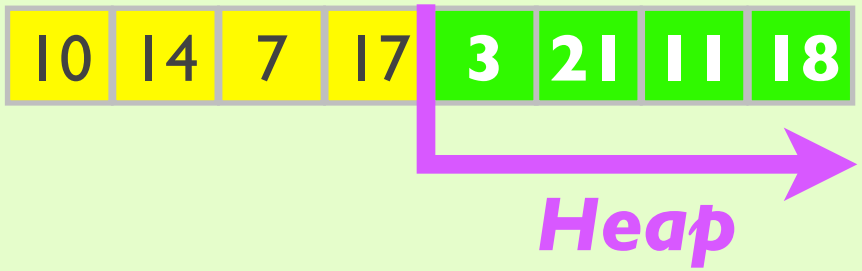
© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 17 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

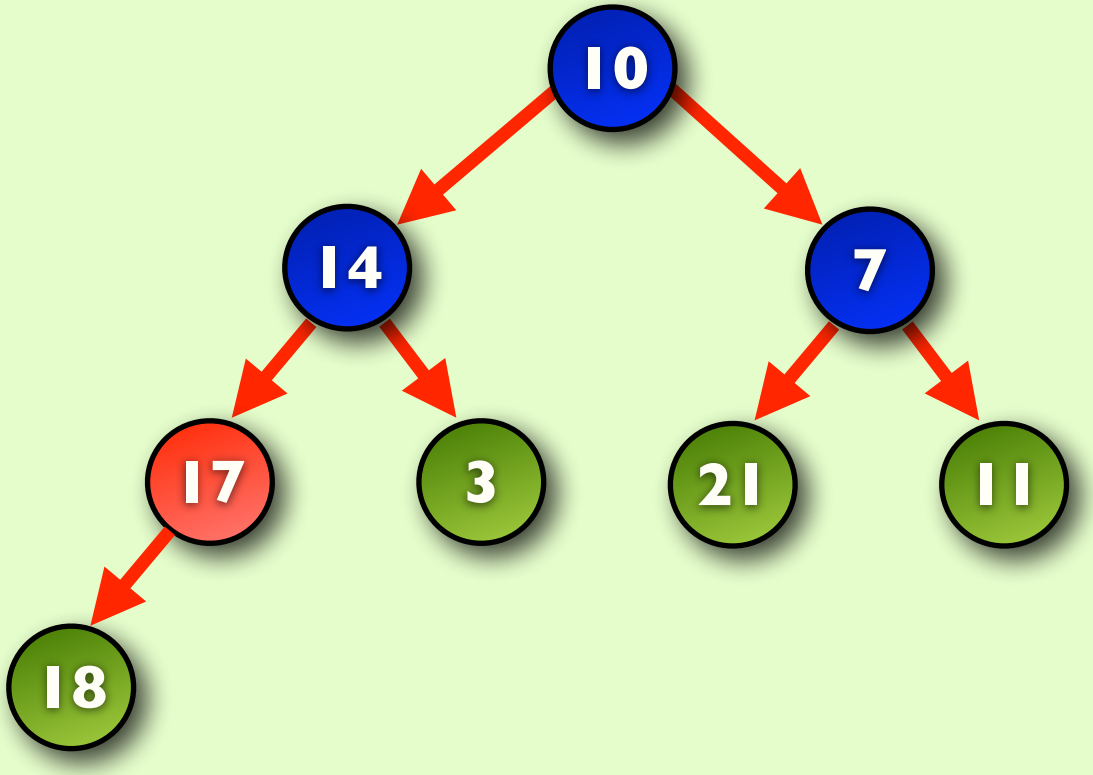


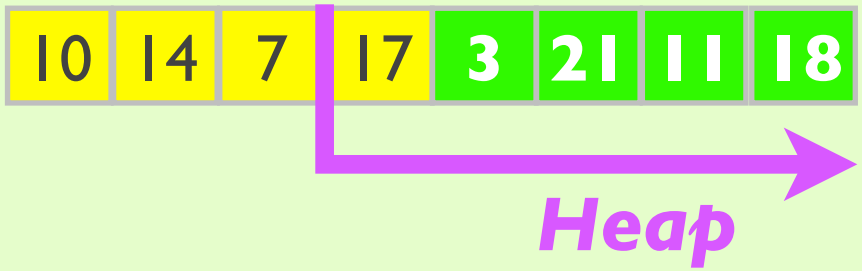


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 17 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

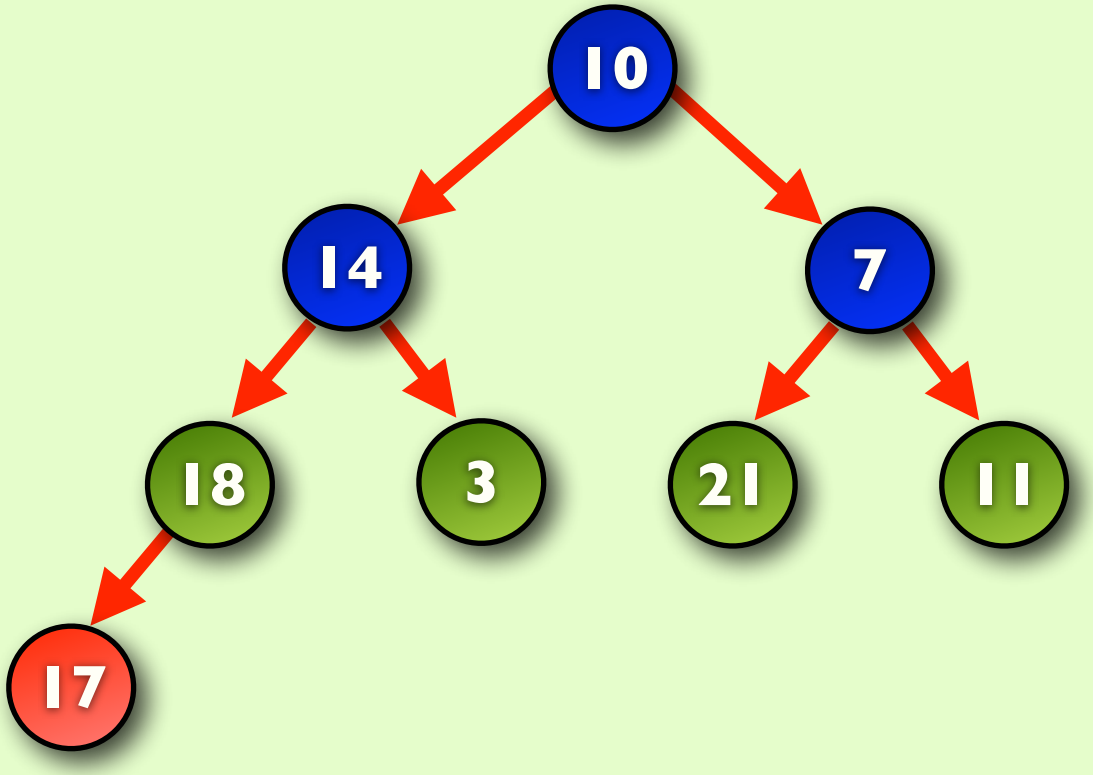


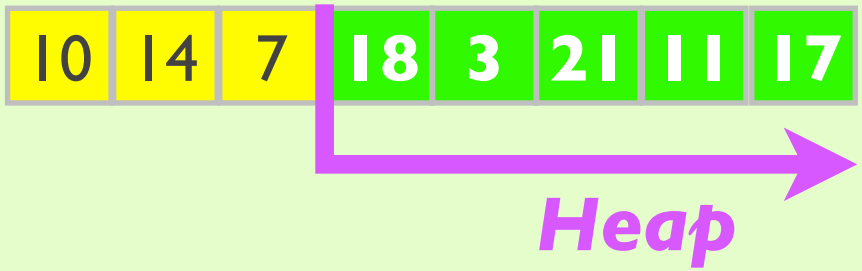


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 17 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.



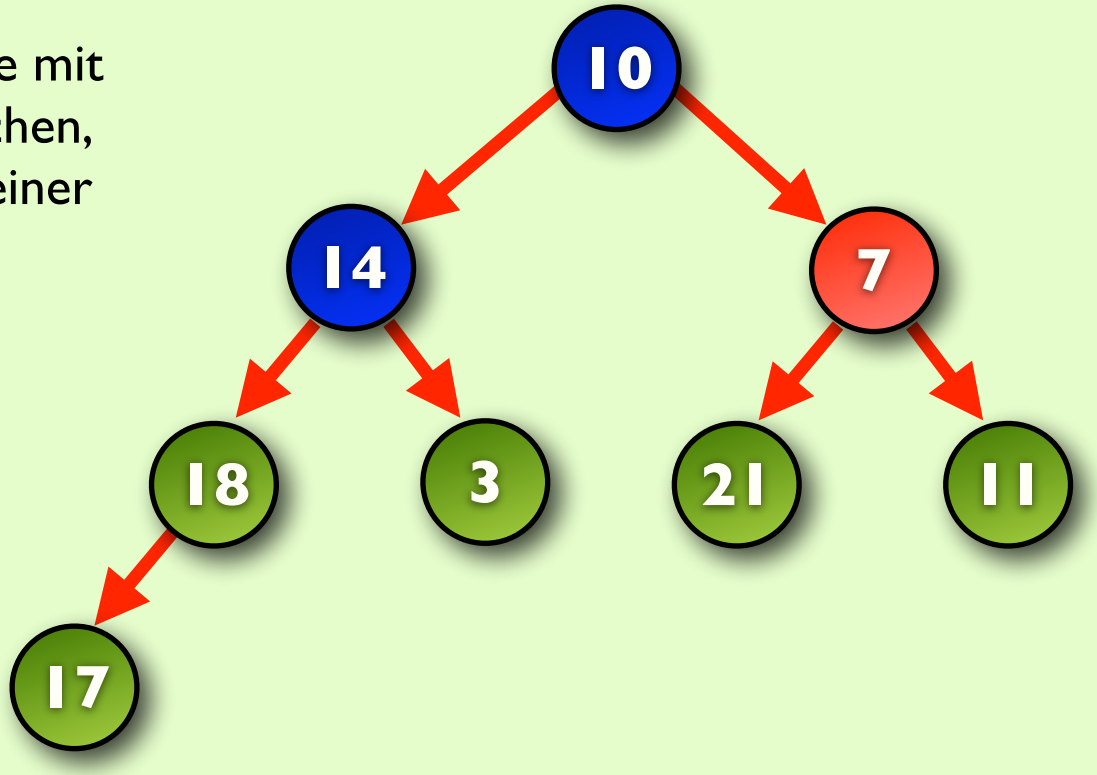


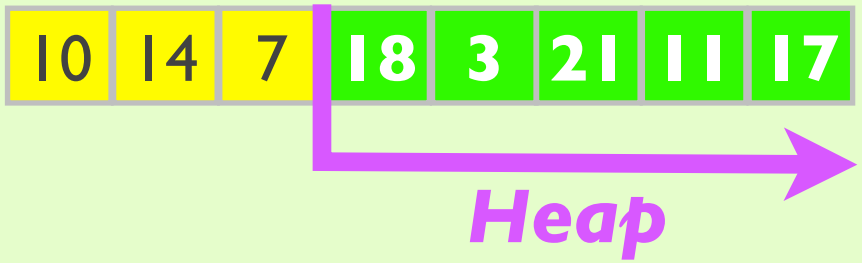
© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 7 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

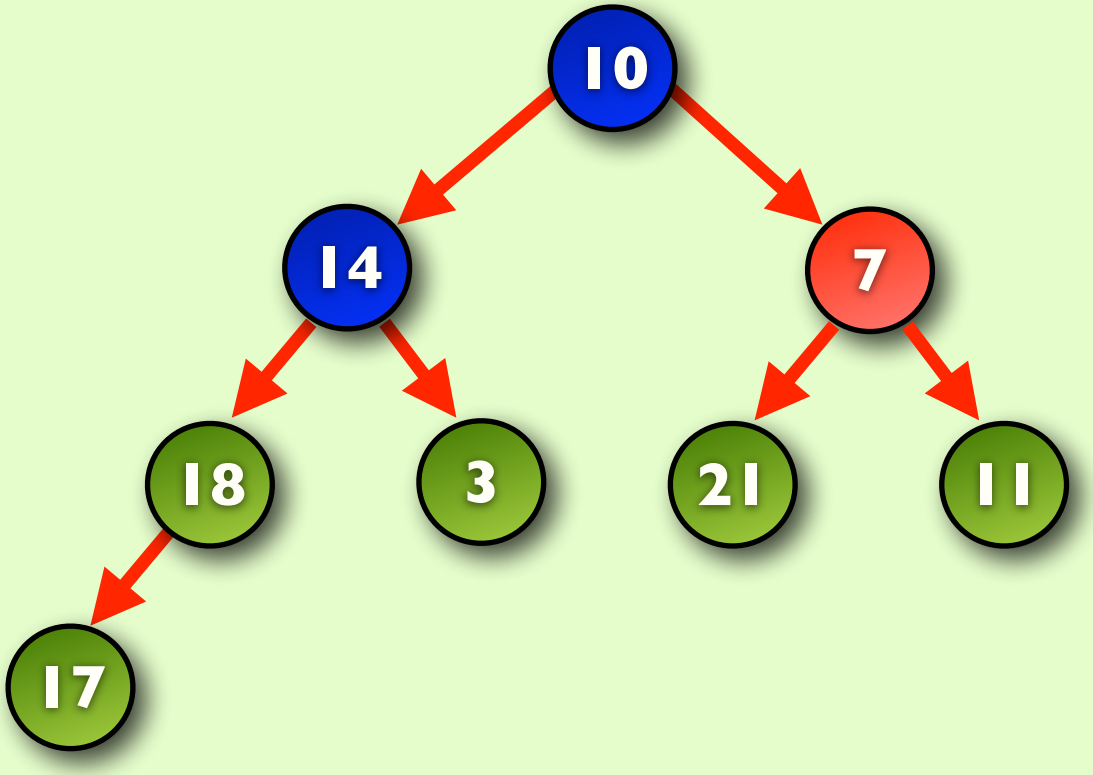




© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

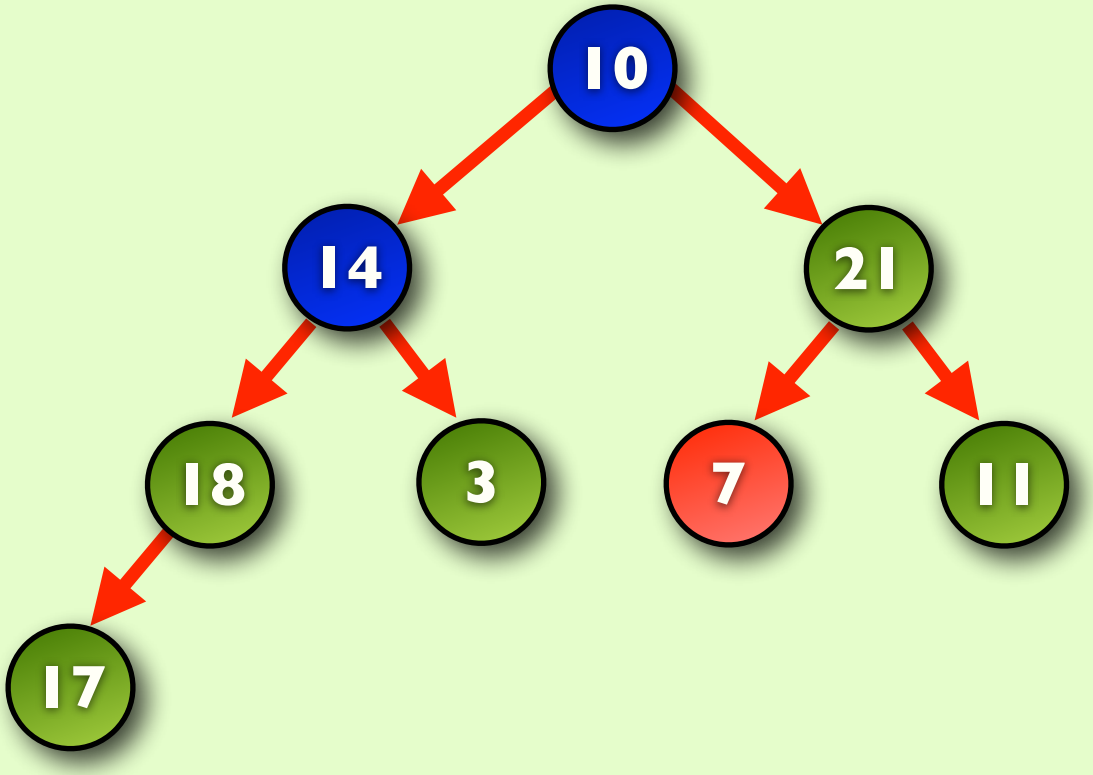
Versickere die 7 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.



© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 7 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

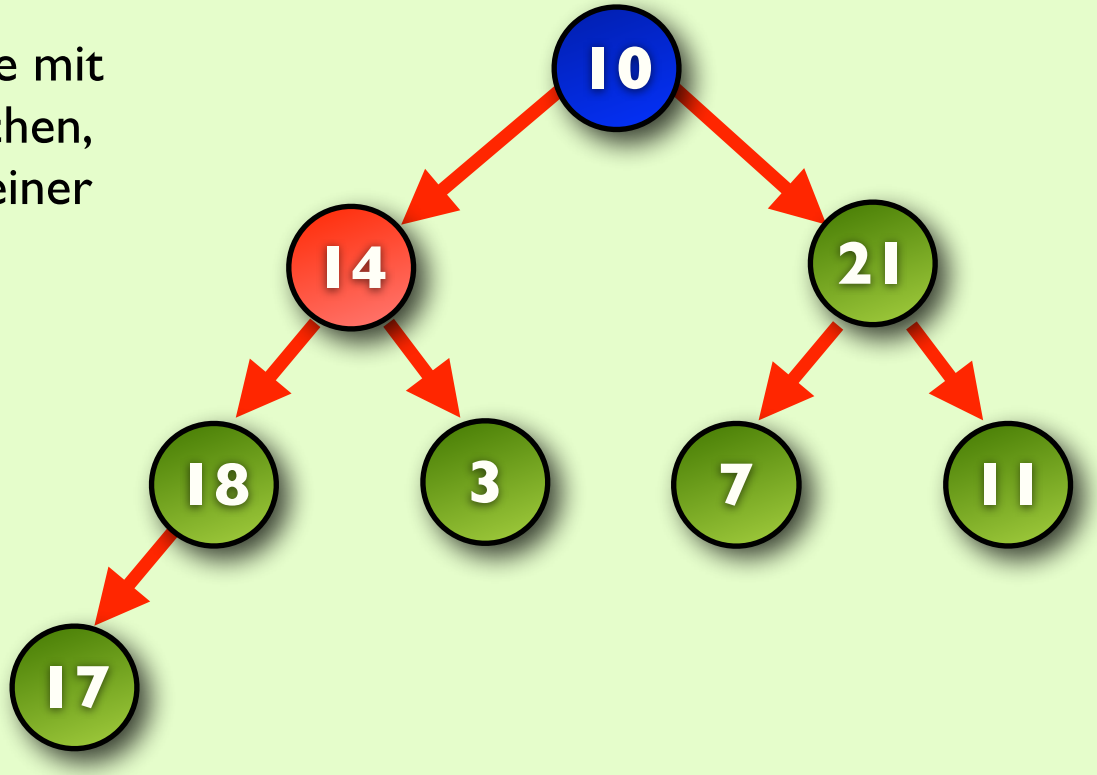


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 14 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

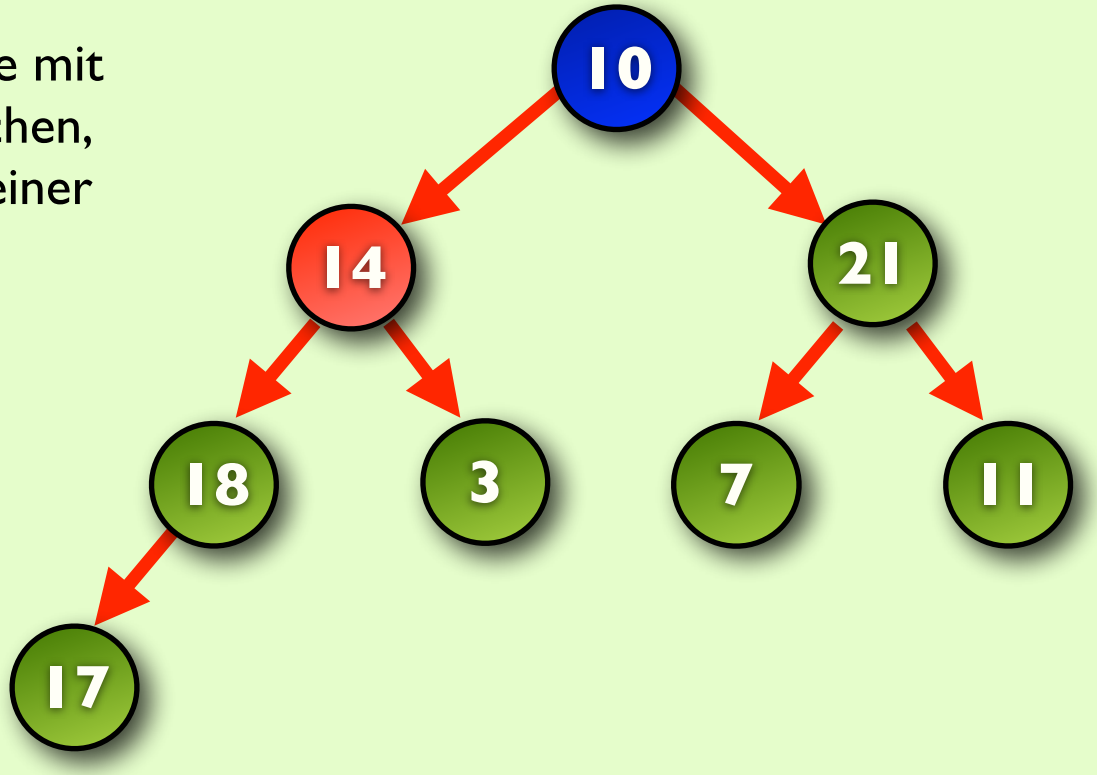


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 14 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

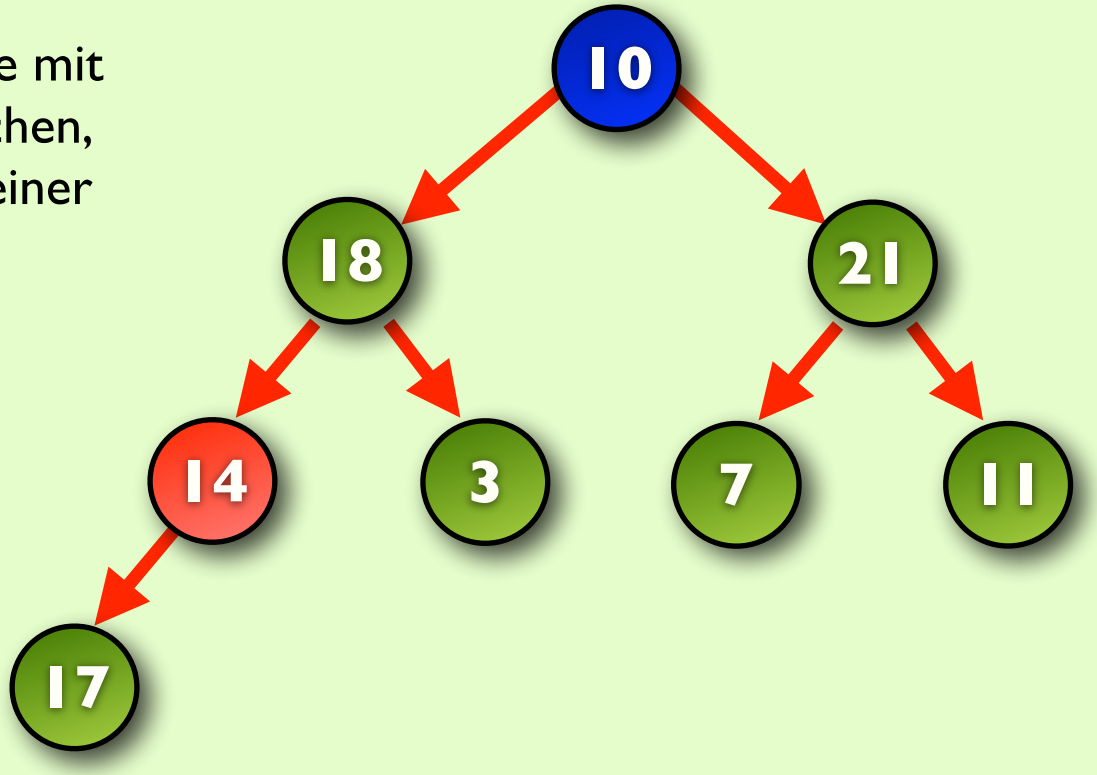


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 14 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

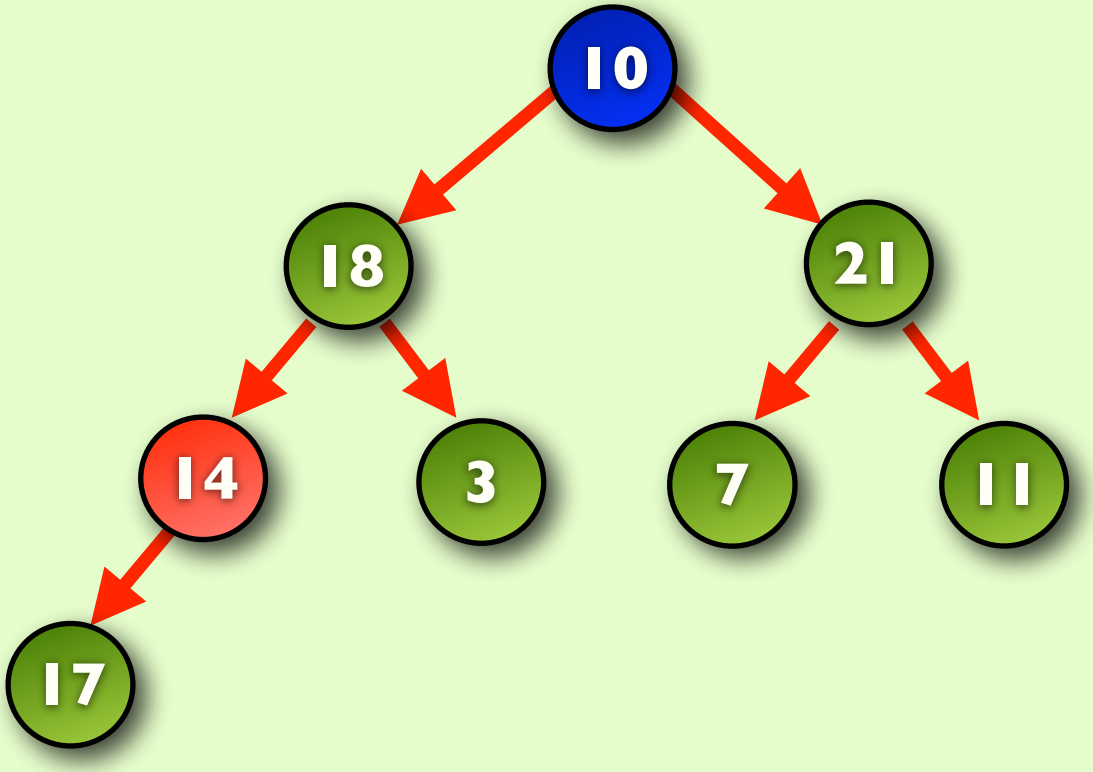


10	18	21	14	3	7	11	17

© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 14 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

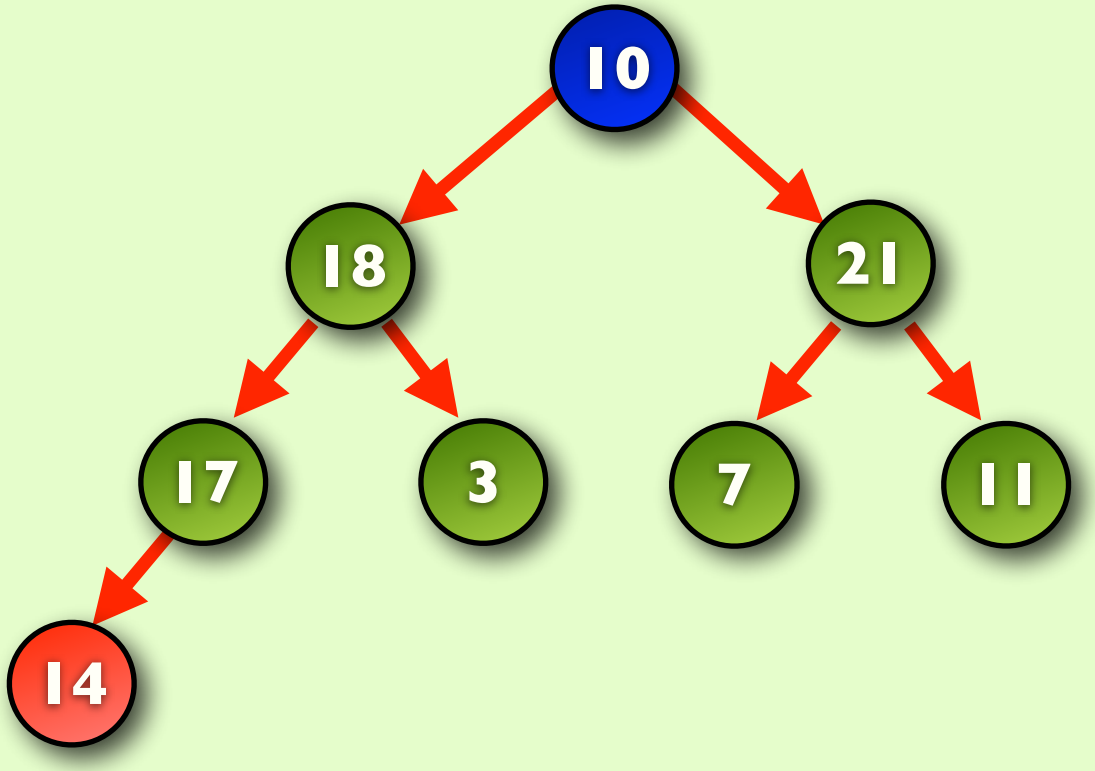


10	18	21	14	3	7	11	17

© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 14 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

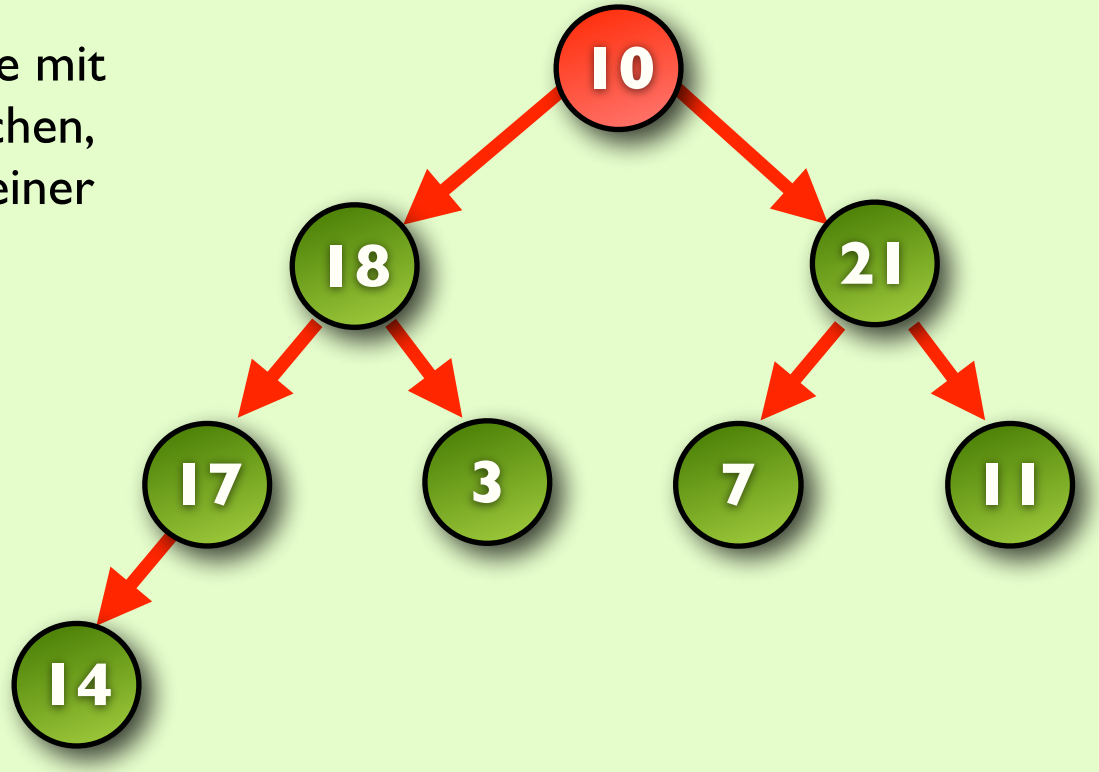


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

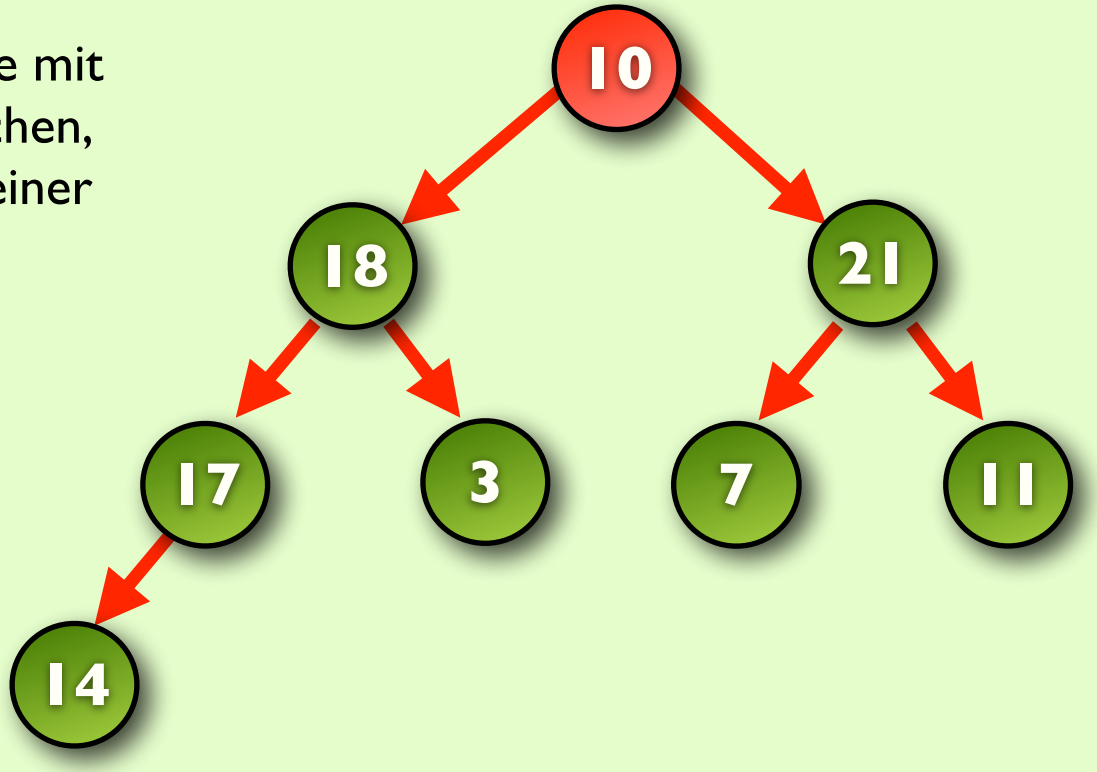


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

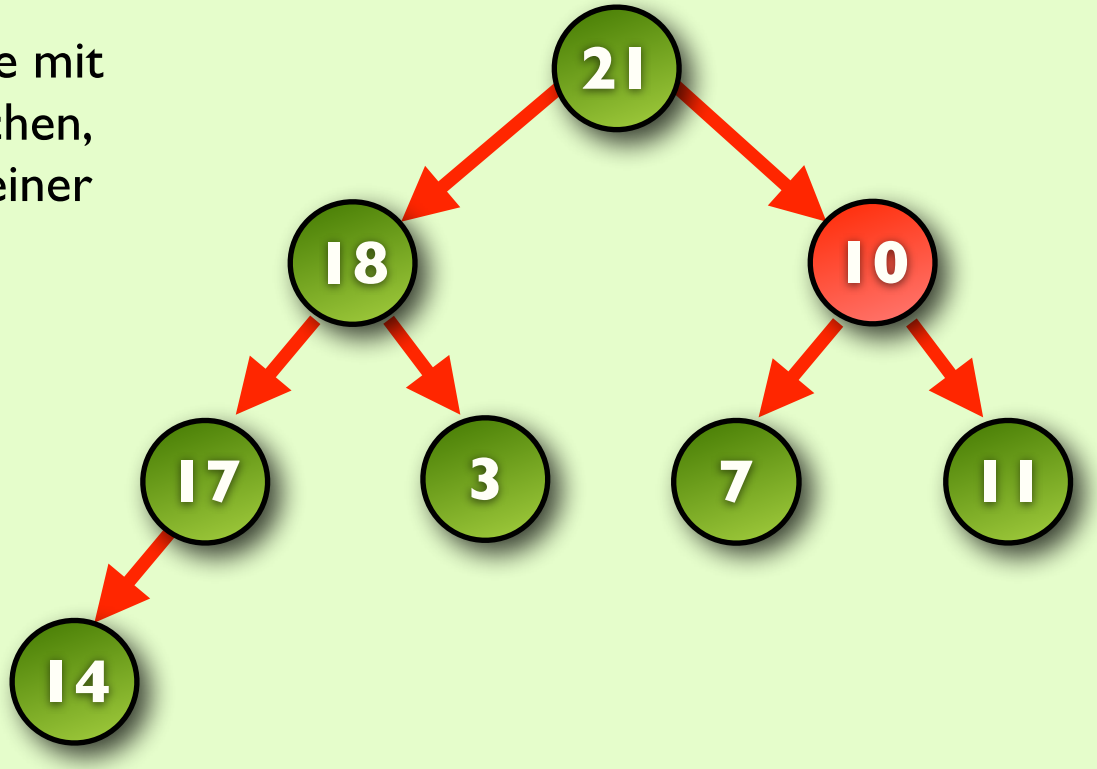


© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

`DownHeap`: Solange mit größtem Kind tauschen, bis beide Kinder kleiner bzw. Blatt erreicht

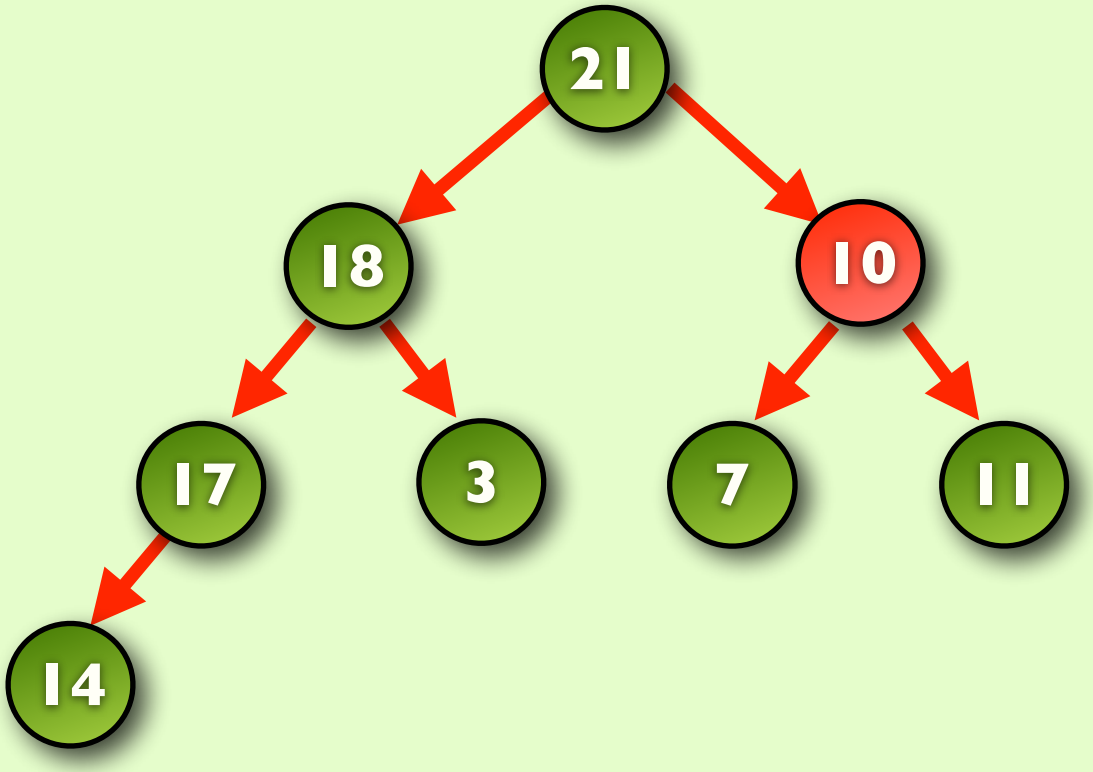


21	18	10	17	3	7	11	14

© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

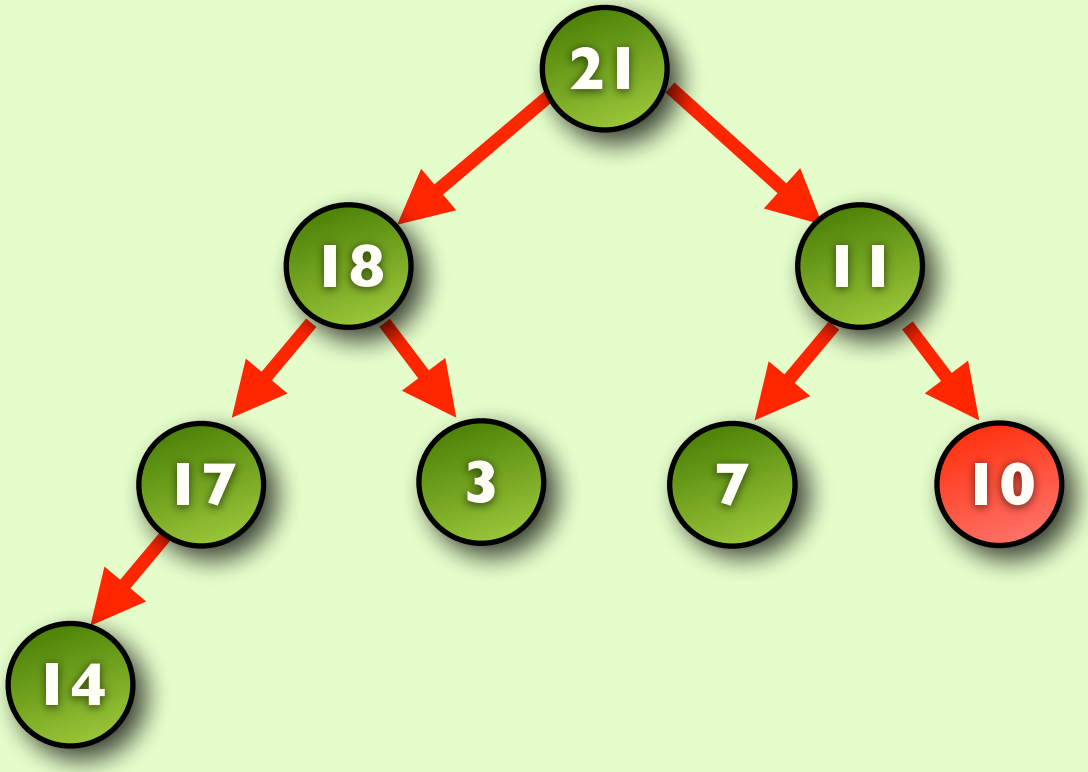


21	18	10	17	3	7	11	14

© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.

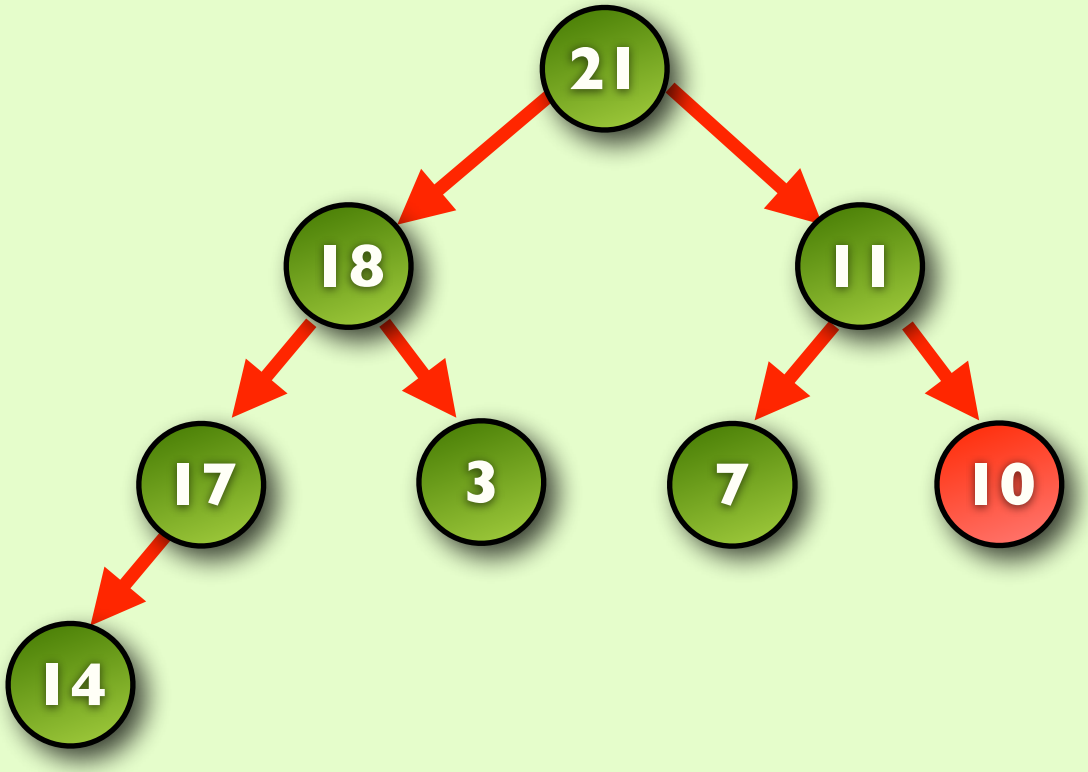


21	18	11	17	3	7	10	14

© 2006-2023 Jörg Striegnitz

Aufbau eines Heap

Versickere die 10 - die Erste Position (von rechts gesehen), ab der die Heap-Eigenschaft verletzt ist.



© 2006-2023 Jörg Striegnitz

Implementierung von DownHeap (JAVA)

```
// i: Index zu versenkendes Element - size: Heap-Grenze
static void DownHeap(int i,int size,int feld[]) {
    int v,j;
    v = feld[i]; // Element merken, dass wir "runterschicken"
    while (true) {
      // Für i >= (size+1)/2 ist die Heap-Eigenschaft erfüllt
      // und es ist nichts zu tun.
      if (i < (size+1)/2) {
        j = 2*i+1;                    // Berechne linken Sohn
        if ((j<size) &&               // existiert ein rechter Sohn
            (feld[j]<feld[j+1]))      // und ist dieser größer?
          j=j+1;                      // ja: dann tauschen wir mit
                                      // diesem
        if (feld[j]<=v) break;        // Beide Söhne kleiner - fertig!
        feld[i]=feld[j];              // Größeren Wert nach oben
        i=j;                          // und von Position j weitersuchen
      } else break;                   // Blatt erreicht
    }
    feld[i] = v;                      // Element an seine Ziel-Position
}
```


© 2006-2023 Jörg Striegnitz

HeapSort: Implementierung (JAVA)

```
public static void HeapSort(int feld[]) {
    // Heap aufbauen
    for (int i=(feld.length/2)-1 ; i>=0 ; --i)
      DownHeap(i,feld.length-1,feld);
    int k=feld.length-1;  // Zielposition für Maximum
    do {
      int t = feld[0];    // Wurzel-Element (Maximum) merken
      feld[0]=feld[k];    // Letztes Element zur Wurzel machen
      feld[k]=t;          // Größtes Element ans Ende
      k=k-1;              // Heap um Eins verkürzen
      DownHeap(0,k,feld); // Heap-Eigenschaft wiederherstellen,
                          // indem wir neue Wurzel "nach unten"
                          // schicken
    } while (k>=0);
  }
```


© 2006-2023 Jörg Striegnitz

Vorüberlegungen zur Komplexitätsanalyse

- **Ein Feld mit N Elementen entspricht einem Baum der Höhe h, wobei**

$$2^h \leq N < 2^{h+1}$$

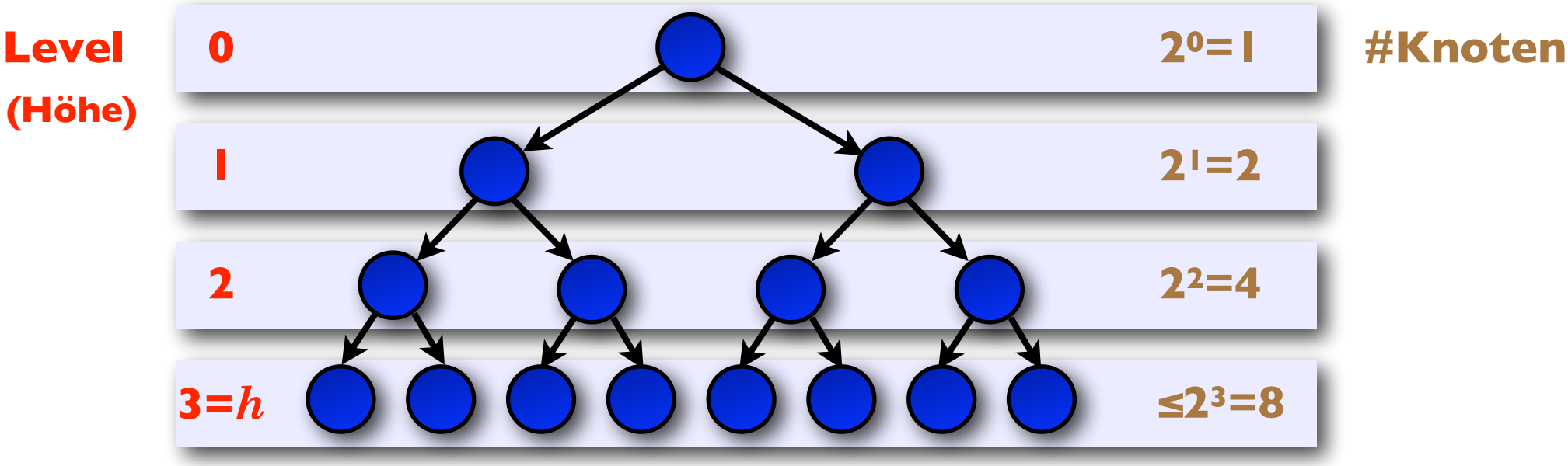
© 2006-2023 Jörg Striegnitz

Vorüberlegungen zur Komplexitätsanalyse

- Ein Feld mit N Elementen entspricht einem Baum der Höhe h, wobei

$$2^h \leq N < 2^{h+1}$$

- Auf Level k gibt es 2^k Elemente



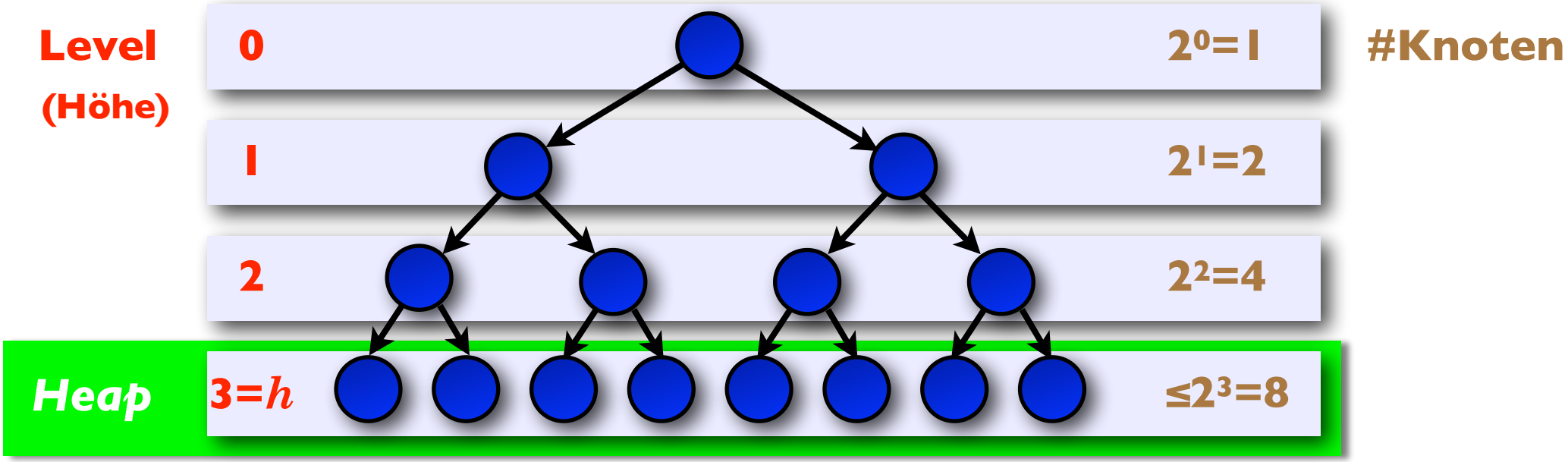
© 2006-2023 Jörg Striegnitz

Vorüberlegungen zur Komplexitätsanalyse

- Ein Feld mit N Elementen entspricht einem Baum der Höhe h, wobei

$$2^h \leq N < 2^{h+1}$$

- Auf Level k gibt es 2^k Elemente



- Wir führen DownHeap für Elemente der Level $(h\text{-}1),(h\text{-}2),\ldots,0$ durch

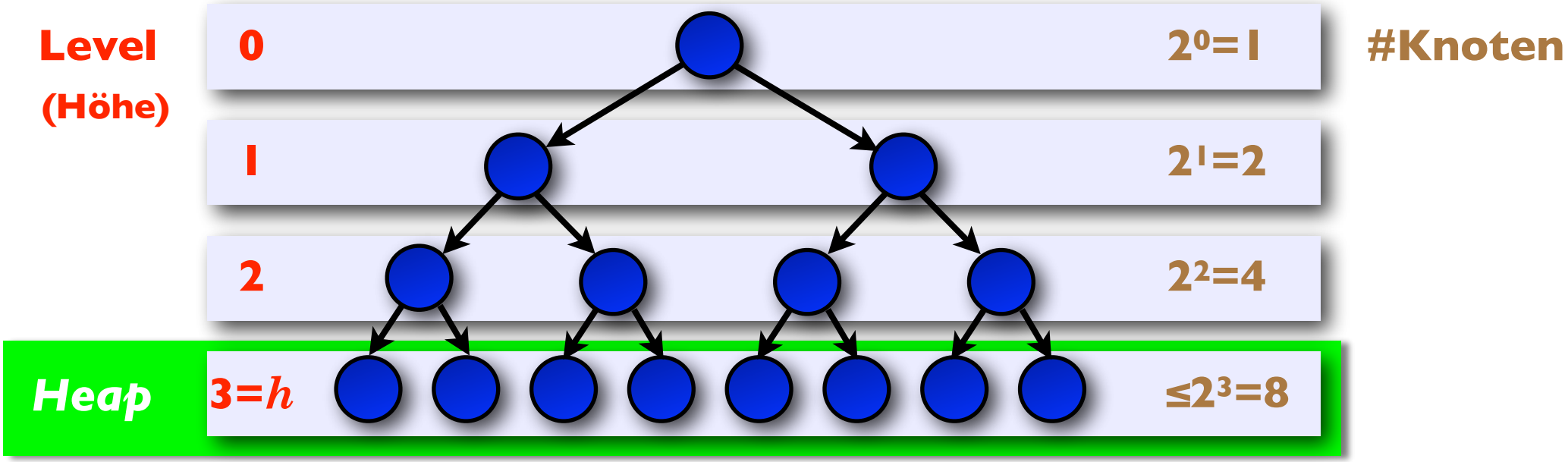
© 2006-2023 Jörg Striegnitz

Vorüberlegungen zur Komplexitätsanalyse

- **Ein Feld mit N Elementen entspricht einem Baum der Höhe h, wobei**

$$2^h \leq N < 2^{h+1}$$

- **Auf Level k gibt es 2^k Elemente**



- **Wir führen DownHeap für Elemente der Level $(h\text{-}1),(h\text{-}2),\ldots,0$ durch**

- **Der maximale Aufwand beim Sinken eines Elementes von Level k ist proportional zu $h\text{-}k$** (# Vergleiche und # Vertauschungen verhalten sich hier gleich!)
  Element kann maximal auf Level h sinken.

© 2006-2023 Jörg Striegnitz

`Heapsort`: Komplexitätsanalyse I

- Wir betrachten ein Feld der Größe N mit $2^h \leq N < 2^{h+1}$ (Baum mit Höhe h)
- **Heap-Aufbau**
  - Wir führen `DownHeap` für Elemente auf Level k=0,1, ... , (h-1) durch
  - Pro Level, den Element sinkt, werden maximal zwei Schlüsselvergleiche benötigt (siehe `DownHeap(i,feld.length-1,feld)`):
    1. Vergleich: `if feld[i]<=v` ...
       Wird bei jedem Durchlauf der `while`-Schleife ausgeführt.
    2. Vergleich: `if feld[j]<feld[j+1]` ...
       Wird nur ausgeführt, wenn der rechte Sohn existiert.
  - Ein Element auf Level k kann maximal (h-k) Level nach unten sinken;
  - 2^k Elemente auf Level k

  - Für die Zahl der Vergleiche ergibt sich folgende obere Schranke

$$\sum_{k=0}^{h-1} \underbrace{2}_{\#\text{ Vergleiche}} \cdot \underbrace{(h-k)}_{\#\text{ Level abwärts}} \cdot \underbrace{2^k}_{\#\text{ Knoten in Level } k}$$

© 2006-2023 Jörg Striegnitz

Heapsort: Komplexitätsanalyse II

- **Abschätzung der oberen Schranke für Anzahl Vergleiche:**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= \sum_{k=0}^{h-1} 2^{k+1} \cdot (h-k) \\
&\quad \text{(* Ausklammern von } 2^h \text{ *)} \\
&= 2^h \cdot \sum_{k=0}^{h-1} 2^{k+1-h} \cdot (h-k) \\
&\quad \text{(* Siehe nächste Folie ... *)} \\
&\leq 2^h \cdot \sum_{k=0}^{h-1} k \cdot 2^{-k} \\
&= 2^h \cdot \sum_{k=0}^{h-1} \frac{k}{2^k} = 2^h \cdot \underbrace{\sum_{k=0}^{h-1} k \cdot \left(\frac{1}{2}\right)^k}_{<2} \\
&< 2^{h+1} \\
&\leq 2 \cdot N
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

Heapsort: Komplexitätsanalyse II

- **Abschätzung der oberen Schranke für Anzahl Vergleiche:**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= \sum_{k=0}^{h-1} 2^{k+1} \cdot (h-k) \\
&\quad \text{(* Ausklammern von } 2^h \text{ *)} \\
&= 2^h \cdot \sum_{k=0}^{h-1} 2^{k+1-h} \cdot (h-k) \\
&\quad \text{(* Siehe nächste Folie ... *)} \\
&\leq 2^h \cdot \sum_{k=0}^{h-1} k \cdot 2^{-k} \\
&= 2^h \cdot \sum_{k=0}^{h-1} \frac{k}{2^k} = 2^h \cdot \underbrace{\sum_{k=0}^{h-1} k \cdot \left(\frac{1}{2}\right)^k}_{<2} \\
&< 2^{h+1} \\
&\leq 2 \cdot N
\end{aligned}
$$

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \quad \text{falls } |x| < 1$$

Für $x = \frac{1}{2}$:

$$\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2$$

© 2006-2023 Jörg Striegnitz

Heapsort: Komplexitätsanalyse II

- **Abschätzung der oberen Schranke für Anzahl Vergleiche:**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= \sum_{k=0}^{h-1} 2^{k+1} \cdot (h-k) \\
&\quad \text{(* Ausklammern von } 2^h \text{ *)} \\
&= 2^h \cdot \sum_{k=0}^{h-1} 2^{k+1-h} \cdot (h-k) \\
&\quad \text{(* Siehe nächste Folie ... *)} \\
&\leq 2^h \cdot \sum_{k=0}^{h-1} k \cdot 2^{-k} \\
&= 2^h \cdot \sum_{k=0}^{h-1} \frac{k}{2^k} = 2^h \cdot \underbrace{\sum_{k=0}^{h-1} k \cdot \left(\frac{1}{2}\right)^k}_{<2} \\
&< 2^{h+1} \\
&\leq 2 \cdot N
\end{aligned}
$$

$\boxed{O(N)}$

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \quad \text{falls } |x| < 1$$

Für $x = \frac{1}{2}$:

$$\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2$$

© 2006-2023 Jörg Striegnitz

Zwischenrechnung ...

- **Umorganisieren der Reihe ...**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2^{k+1-h} \cdot (h-k) &= \underbrace{2^{1-h}(h-0)}_{k=0} + \underbrace{2^{2-h}(h-1)}_{k=1} + \cdots + \underbrace{2^{-1}(h-(h-2))}_{k=h-2} + \underbrace{2^{0}(h-(h-1))}_{k=h-1} \\
&= \sum_{k=0}^{h-1} 2^{-k}(h-(h-k-1)) \\
&= \sum_{k=0}^{h-1} 2^{-k} \cdot h - 2^{-k} \cdot h + 2^{-k} \cdot k + 2^{-k} \\
&= \sum_{k=0}^{h-1} 2^{-k} \cdot k + \sum_{k=0}^{h-1} 2^{-k} \\
&\leq \sum_{k=0}^{h-1} 2^{-k} \cdot k
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

Exkurs

Einschub:
Exakte Berechnung der Anzahl der Vergleiche

© 2006-2023 Jörg Striegnitz

Heapsort: Exakte Analyse der Vergleiche I

- **Wir teilen die Summe auf und entdecken die Geometrische Reihe:**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= 2 \cdot \left[h \cdot \underbrace{\sum_{k=0}^{h-1} 2^k}_{\text{Geometrische Reihe}} - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot \frac{2^{(h-1)+1} - 1}{2-1} - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot (2^h - 1) - \sum_{k=0}^{h-1} k \cdot 2^k \right]
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

Heapsort: Exakte Analyse der Vergleiche I

- **Wir teilen die Summe auf und entdecken die Geometrische Reihe:**

$$
\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= 2 \cdot \left[h \cdot \underbrace{\sum_{k=0}^{h-1} 2^k}_{\text{Geometrische Reihe}} - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot \frac{2^{(h-1)+1} - 1}{2-1} - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot (2^h - 1) - \boxed{\sum_{k=0}^{h-1} k \cdot 2^k} \right]
\end{aligned}
$$

Auflösung auf nächster Folie ...

© 2006-2023 Jörg Striegnitz

Heapsort: Exakte Analyse der Vergleiche II

(* Geometrische Reihe *)

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

(* Differenzieren und mit x multiplizieren *)

$$\Leftrightarrow \quad x \cdot \sum_{k=0}^{n} k \cdot x^{k-1} = x \cdot \left(\frac{x^{n+1}-1}{x-1} \frac{d}{dx} \right)$$

$$\Leftrightarrow \quad \sum_{k=0}^{n} k \cdot x^k = x \cdot \frac{(n+1)x^n \cdot (x-1) - (x^{n+1}-1)}{(x-1)^2}$$

$$\Leftrightarrow \quad \sum_{k=0}^{n} k \cdot x^k = x \cdot \frac{(n+1) \cdot x^{n^+1} - (n+1) \cdot x^n - x^{n+1} + 1}{(x-1)^2}$$

$$\Leftrightarrow \quad \sum_{k=0}^{n} k \cdot x^k = x \cdot \frac{n \cdot x^{n^+1} - (n+1) \cdot x^n + 1}{(x-1)^2}$$

$$\Leftrightarrow \quad \sum_{k=0}^{n} k \cdot x^k = \frac{n \cdot x^{n^+2} - (n+1) \cdot x^{n+1} + x}{(x-1)^2}$$

© 2006-2023 Jörg Striegnitz

Heapsort: Exakte Analyse der Vergleiche III

- **Setzen wir unsere konkreten Werte ein:**

$$\sum_{k=0}^{n} k \cdot x^k = \frac{n \cdot x^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}$$

(* Setze $x = 2$ und $n = h - 1$ *)

$$\begin{aligned}
\sum_{k=0}^{h-1} k \cdot 2^k &= \frac{(h-1) \cdot 2^{(h-1)+2} - ((h-1)+1) \cdot 2^{(h-1)+1} + 2}{(2-1)^2} \\
&= (h-1) \cdot 2^{h+1} - h \cdot 2^h + 2 \\
&= 2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2
\end{aligned}$$

- **Insgesamt ergibt sich nun:**

$$\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= 2 \cdot \left[h \cdot (2^h - 1) - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot (2^h - 1) - \left(2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2\right) \right] \\
&= 2 \cdot \left[h \cdot 2^h - h - 2 \cdot (h-1) \cdot 2^h + h \cdot 2^h - 2 \right] \\
&= 2 \cdot \left[\underbrace{(h - 2 \cdot h + 2 + h)}_{=2} \cdot 2^h - h - 2 \right] \\
&= 2 \cdot \left[2^{h+1} - h - 2 \right] \\
&= 2^{h+1} - 2 \cdot (h+2) < N - 2 \cdot (h+2) < N
\end{aligned}$$

© 2006-2023 Jörg Striegnitz

Heapsort: Exakte Analyse der Vergleiche III

- **Setzen wir unsere konkreten Werte ein:**

$$\sum_{k=0}^{n} k \cdot x^k = \frac{n \cdot x^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}$$

(* Setze $x = 2$ und $n = h - 1$ *)

$$\begin{aligned}
\sum_{k=0}^{h-1} k \cdot 2^k &= \frac{(h-1) \cdot 2^{(h-1)+2} - ((h-1)+1) \cdot 2^{(h-1)+1} + 2}{(2-1)^2} \\
&= (h-1) \cdot 2^{h+1} - h \cdot 2^h + 2 \\
&= 2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2
\end{aligned}$$

- **Insgesamt ergibt sich nun:**

$$\begin{aligned}
\sum_{k=0}^{h-1} 2 \cdot (h-k) \cdot 2^k &= 2 \cdot \left[h \cdot (2^h - 1) - \sum_{k=0}^{h-1} k \cdot 2^k \right] \\
&= 2 \cdot \left[h \cdot (2^h - 1) - \left(2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2\right) \right] \\
&= 2 \cdot \left[h \cdot 2^h - h - 2 \cdot (h-1) \cdot 2^h + h \cdot 2^h - 2 \right] \\
&= 2 \cdot \left[\underbrace{(h - 2 \cdot h + 2 + h)}_{=2} \cdot 2^h - h - 2 \right] \\
&= 2 \cdot \left[2^{h+1} - h - 2 \right] \\
&= 2^{h+1} - 2 \cdot (h+2) < N - 2 \cdot (h+2) < N
\end{aligned}$$

$\boxed{O(N)}$

© 2006-2023 Jörg Striegnitz

`Heapsort`: Komplexitätsanalyse III

- **Sortierphase**
  - Nach Heap-Aufbau muss endgültige Ordnung auf Feld hergestellt werden
  - Dazu wird Knoten der Tiefe k auf die Wurzel gesetzt und mittels `DownHeap` maximal um k Level sinken (**Achtung**: nicht h-Level, da Heap verkürzt wird!).
  - Pro Level sind maximal 2 Vergleiche notwendig
  - Damit ergibt sich für die Zahl der Vergleiche in der Sortierphase:

$$
\begin{aligned}
\sum_{k=0}^{h} 2 \cdot k \cdot 2^k &= \left(2 \cdot \sum_{k=0}^{h-1} k \cdot 2^k\right) + 2h \cdot 2^h \\
&= 2 \cdot \left[2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2\right] + 2h \cdot 2^h \\
&= 4h \cdot 2^h - 4 \cdot 2^h - 2h \cdot 2^h + 4 + 2h \cdot 2^h \\
&= 2^{h+1} \cdot (2h-2) + 4
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

`Heapsort`: Komplexitätsanalyse III

- **Sortierphase**
  - Nach Heap-Aufbau muss endgültige Ordnung auf Feld hergestellt werden
  - Dazu wird Knoten der Tiefe k auf die Wurzel gesetzt und mittels `DownHeap` maximal um k Level sinken (**Achtung**: nicht h-Level, da Heap verkürzt wird!).
  - Pro Level sind maximal 2 Vergleiche notwendig
  - Damit ergibt sich für die Zahl der Vergleiche in der Sortierphase:

Es gilt (ggf. Beweis durch vollständige Induktion):

$$\boxed{\sum_{k=0}^{h-1} k \cdot 2^k = 2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2}$$

$$\begin{aligned}
\sum_{k=0}^{h} 2 \cdot k \cdot 2^k &= \left(2 \cdot \sum_{k=0}^{h-1} k \cdot 2^k\right) + 2h \cdot 2^h \\
&= 2 \cdot \left[2 \cdot (h-1) \cdot 2^h - h \cdot 2^h + 2\right] + 2h \cdot 2^h \\
&= 4h \cdot 2^h - 4 \cdot 2^h - 2h \cdot 2^h + 4 + 2h \cdot 2^h \\
&= 2^{h+1} \cdot (2h-2) + 4
\end{aligned}$$

© 2006-2023 Jörg Striegnitz

`Heapsort`: Komplexitätsanalyse IV

- **Gesamtzahl der Vergleiche**
  - Wir betrachten voll besetzten Baum der Höhe h (N=2^{h+1}-1)
  - Für Aufbau des Heap und Sortierphase ergibt sich:

© 2006-2023 Jörg Striegnitz

Heapsort: Komplexitätsanalyse IV

- **Gesamtzahl der Vergleiche**
  - Wir betrachten voll besetzten Baum der Höhe h (N=2^{h+1}-1)
  - Für Aufbau des Heap und Sortierphase ergibt sich:

$$
\begin{aligned}
T(N) &= \underbrace{2^{h+1} - 2 \cdot (h+2)}_{Heapaufbau} + \underbrace{2^{h+1} \cdot (2h-2) + 4}_{Sortierphase} \\
&= 2^{h+1} - 2h - 4 + 2h \cdot 2^{h+1} - 2 \cdot 2^{h+1} + 4 \\
&= 2h \cdot (2^{h+1} - 1) - 2^{h+1} \\
& \quad (*\ N = 2^{h+1} - 1\ *) \\
&= 2h \cdot N - (N+1) \\
& \quad (*\ N = 2^{h+1} - 1 \Leftrightarrow h = \mathrm{ld}(N+1) - 1\ *) \\
&= 2N \cdot (\mathrm{ld}(N+1) - 1) - (N+1) \\
&= 2N \cdot \mathrm{ld}(N+1) - 3N - 1
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

Heapsort: Komplexitätsanalyse IV

- **Gesamtzahl der Vergleiche**
  - Wir betrachten voll besetzten Baum der Höhe h (N=2^{h+1}-1)
  - Für Aufbau des Heap und Sortierphase ergibt sich:

$$
\begin{aligned}
T(N) &= \underbrace{2^{h+1} - 2 \cdot (h+2)}_{Heapaufbau} + \underbrace{2^{h+1} \cdot (2h-2) + 4}_{Sortierphase} \\
&= 2^{h+1} - 2h - 4 + 2h \cdot 2^{h+1} - 2 \cdot 2^{h+1} + 4 \\
&= 2h \cdot (2^{h+1} - 1) - 2^{h+1} \\
& \quad (*\ N = 2^{h+1} - 1\ *) \\
&= 2h \cdot N - (N+1) \\
& \quad (*\ N = 2^{h+1} - 1 \Leftrightarrow h = \mathrm{ld}(N+1) - 1\ *) \\
&= 2N \cdot (\mathrm{ld}(N+1) - 1) - (N+1) \\
&= 2N \cdot \mathrm{ld}(N+1) - 3N - 1
\end{aligned}
$$

$$\boxed{O(N \cdot \mathrm{ld}\ N)}$$

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- **4.2.3. Merge Sort**

4.3.Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Merge Sort

- **Wurde erstmals 1945 durch John von Neumann vorgestellt**

© 2006-2023 Jörg Striegnitz

Merge Sort

- **Wurde erstmals 1945 durch John von Neumann vorgestellt**

- **Basiert auf dem Prinzip „Teile und Herrsche“**

© 2006-2023 Jörg Striegnitz

Merge Sort

- **Wurde erstmals 1945 durch John von Neumann vorgestellt**

- **Basiert auf dem Prinzip „Teile und Herrsche“**

- **Prinzip** (Sei $F=x_1,\ldots,x_n$ eine Folge von N Schlüsseln)
  - **Divide**: Zerlege F in zwei gleichgroße Folgen $F_1=x_1,\ldots,x_{\lceil n/2 \rceil}$ und $F_2=x_{\lceil n/2 \rceil+1},\ldots x_N$, rekursiv, bis Teilfolgen der Länge 1 entstehen
  - **Conquer**: trivial: Folgen der Länge 1 sind sortiert!
  - **Merge**: **Verschmelze** sortierte Teilfolgen sukzessive zu sortierten Gesamtfolgen

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- **Voraussetzung:** zwei aufsteigend sortierte Listen F_1 und F_2
- Zwei Zeiger p und q auf das jeweils kleinste Element von F_1 und F_2
- Ergebnisfeld R

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- Vergleiche $F_1[p]$ und $F_2[q]$

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- Übernimm den kleineren Schlüsselwert (hier 1) in das Zielfeld

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- und erhöhe den entsprechenden Zeiger (hier p) um 1

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- Vergleiche $F_1[p]$ und $F_2[q]$

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- Übernimm den kleineren Schlüsselwert (hier 2) in das Zielfeld

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- und erhöhe den entsprechenden Zeiger (hier q) um 1

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- usw.

© 2006-2023 Jörg Striegnitz

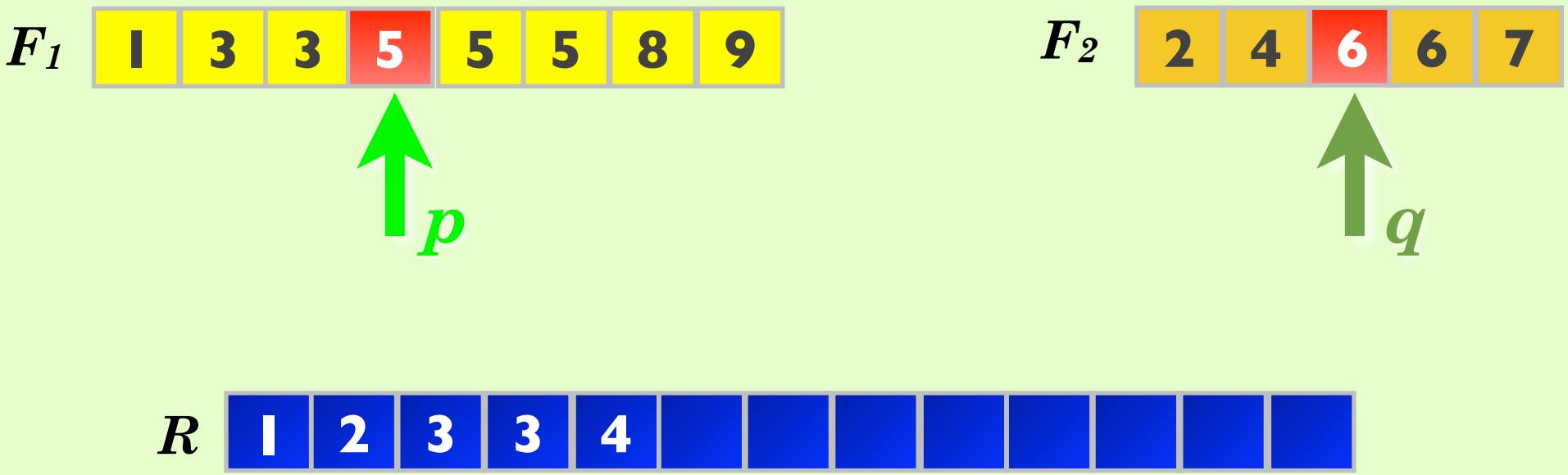
B Verschmelzen zweier sortierter Listen

- usw.

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- usw.



© 2006-2023 Jörg Striegnitz

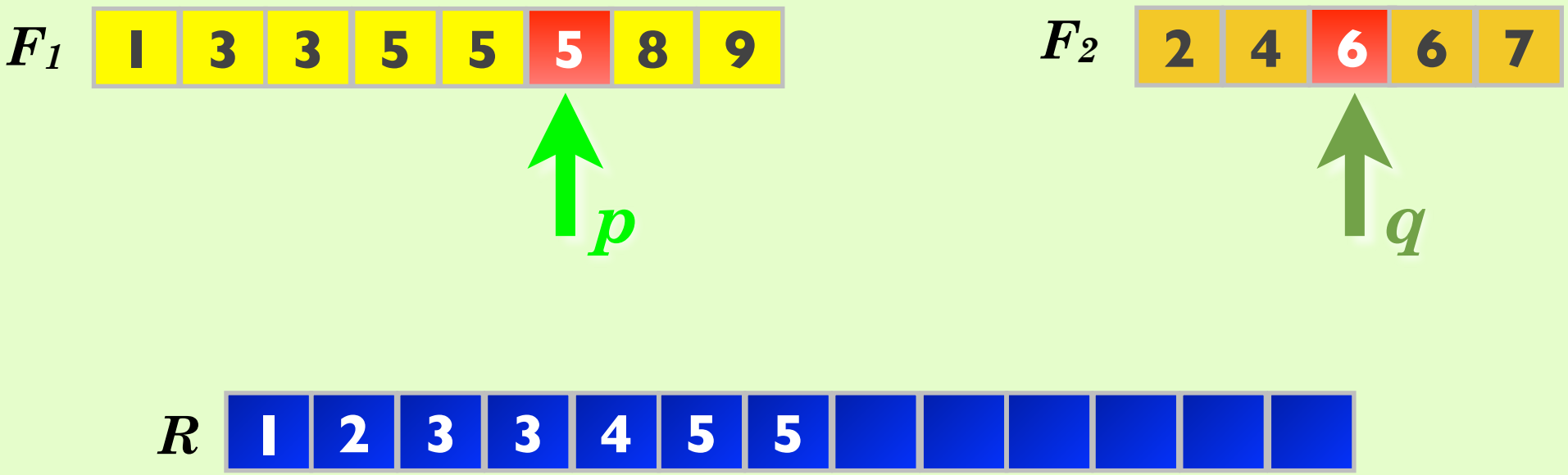
B Verschmelzen zweier sortierter Listen

- usw.

© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

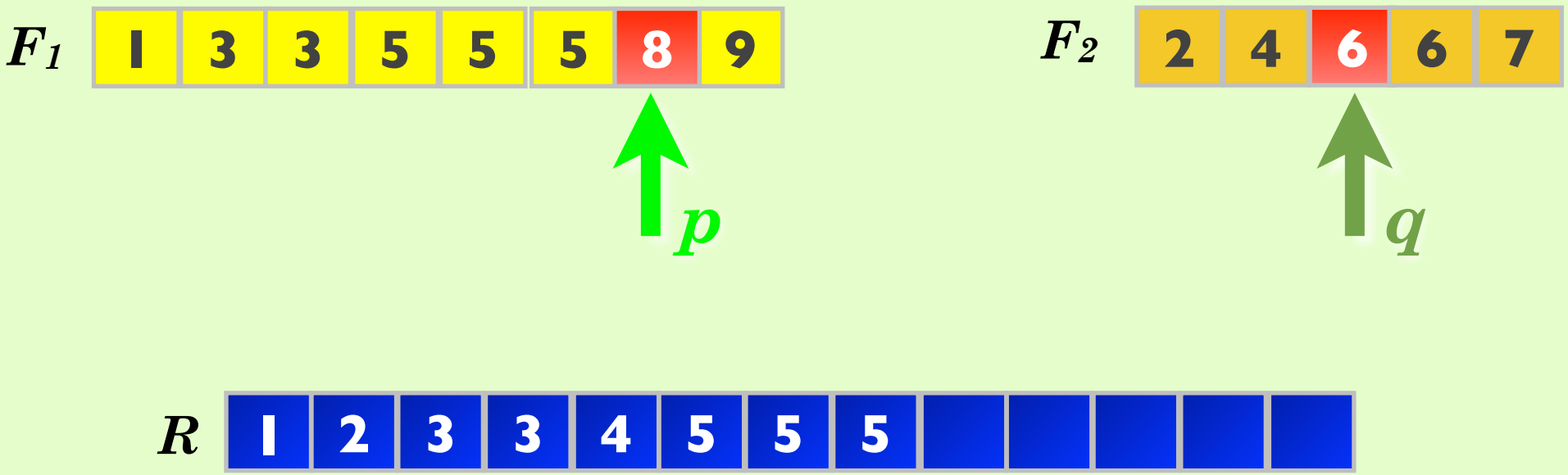
- usw.



© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

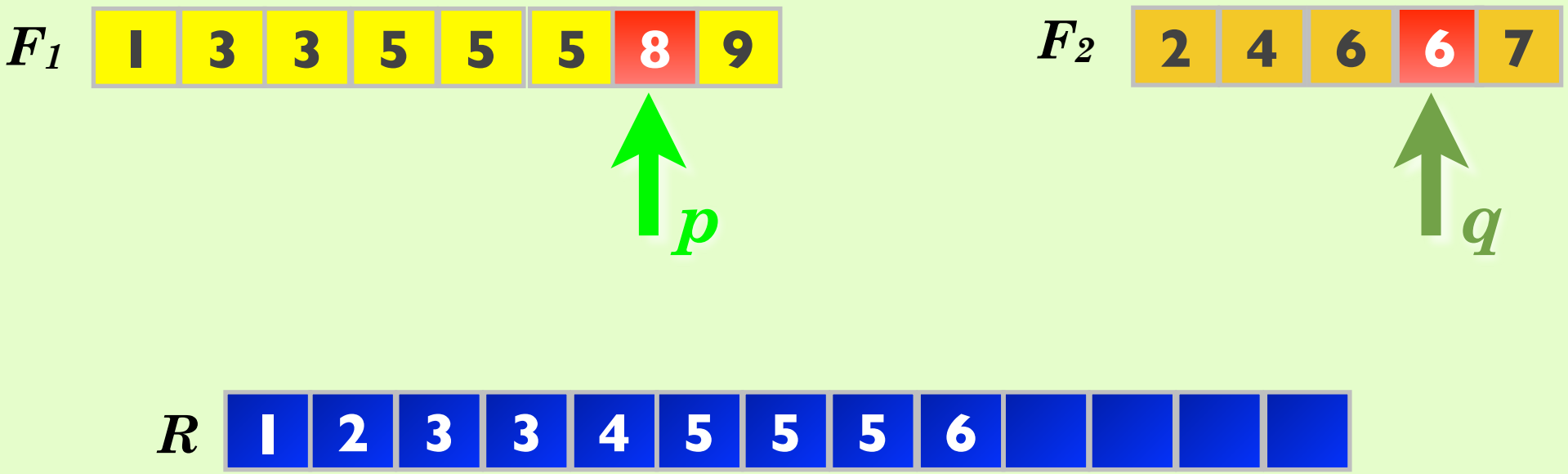
- usw.



© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

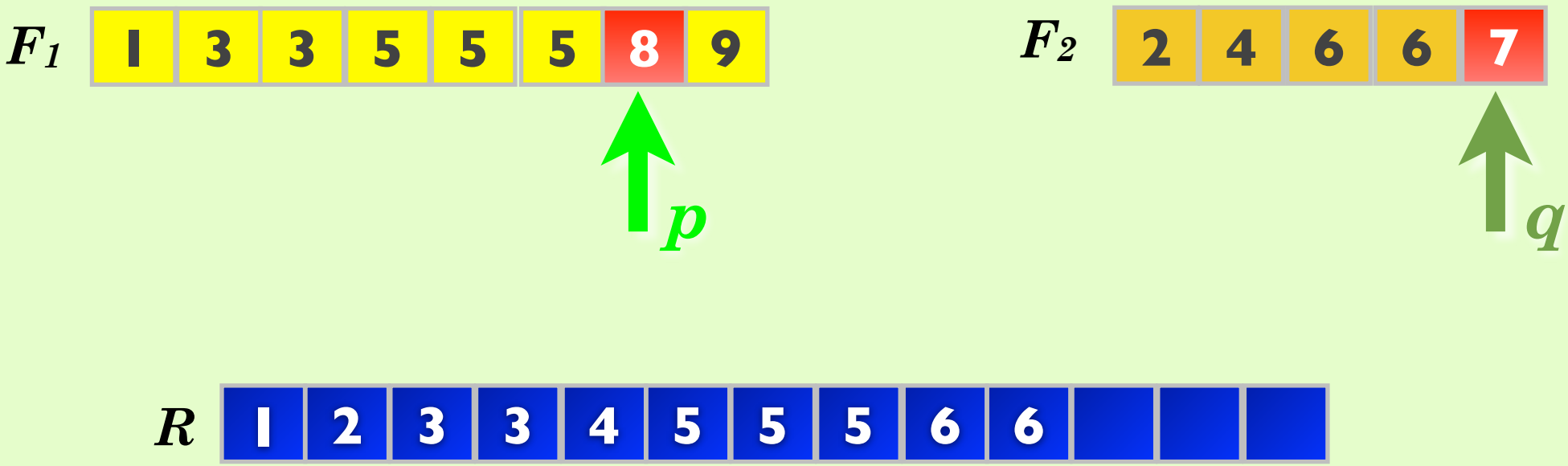
- usw.



© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

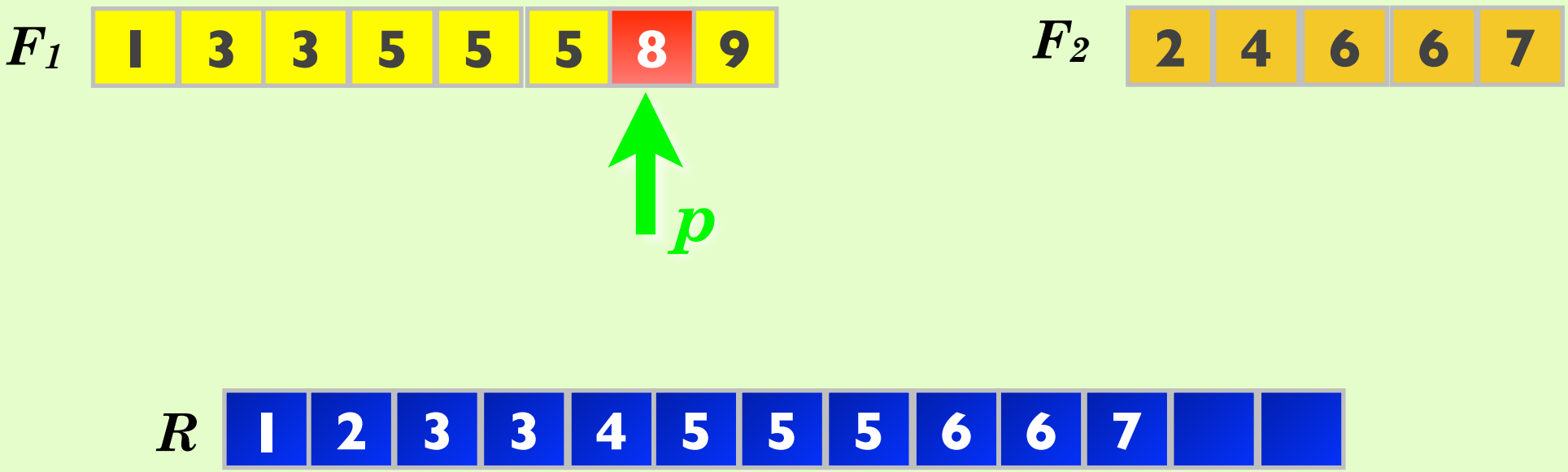
- usw.



© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- ... bis ein Feld vollständig in das Zielfeld kopiert wurde (hier F_2)



© 2006-2023 Jörg Striegnitz

B Verschmelzen zweier sortierter Listen

- dann hänge den Rest des verbleibenden Feldes (hier F_1) an das Ergebnisfeld an

F_1

1	3	3	5	5	5	8	9

F_2

2	4	6	6	7

R

1	2	3	3	4	5	5	5	6	6	7	8	9

© 2006-2023 Jörg Striegnitz

Merge: Implementierung (JAVA)

```
//   Teilfeld 1 : feld[left .. middle]
//   Teilfeld 2:  feld[middle+1 .. right]
static void Merge(int left,int middle, int right, int feld[]) {
  int p=left , q=middle+1, r=0;
  int result[] = new int[right-left+1]; // Ergebnisfeld
  // Solange nicht das Ende eines Feldes erreicht ...
  while ((p <= middle) && (q <= right)) {
    // den jeweils kleineren Wert ins Zielfeld kopieren
    if (feld[p] <= feld[q])
      result[r++] = feld[p++];
    else
      result[r++] = feld[q++];
  }
  // Eventuellen Rest anhängen
  while (p<=middle) result[r++]=feld[p++];
  while (q<=right) result[r++]=feld[q++];
  for (int i=left ; i<=right ; ++i)
    feld[i] = result[i-left];
}
```


© 2006-2023 Jörg Striegnitz

Merge Sort: Implementierung (JAVA)

- **Basierend auf `Merge` kann `Merge Sort` einfach realisiert werden:**

```
static void MergeSort(int l,int r,int feld[]) {
  if (l<r) {  // Gibt's was zu tun?
    int middle = (l+r)/2;          // Mitte berechnen
    MergeSort(l,middle,feld);      // linkes Teilfeld sortieren
    MergeSort(middle+1,r,feld);    // rechtes Teilfeld sortieren
    Merge(l,middle,r,feld);        // Ergebnis zusammenmischen
  }
}

// Überladene Version für vereinfachten Aufruf:
static void MergeSort(int feld[]) {
  MergeSort(0,feld.length-1,feld);
}
```


© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen

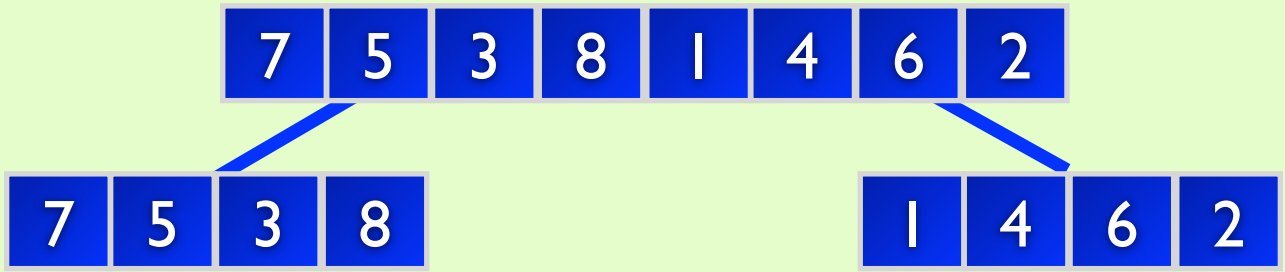
7	5	3	8	1	4	6	2

© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen

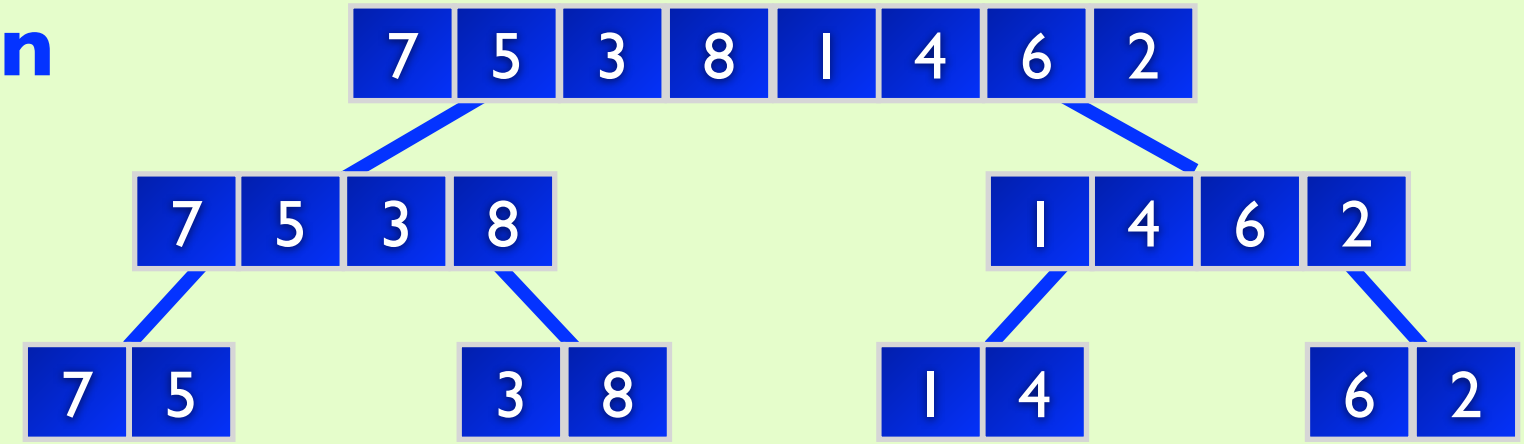


© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen

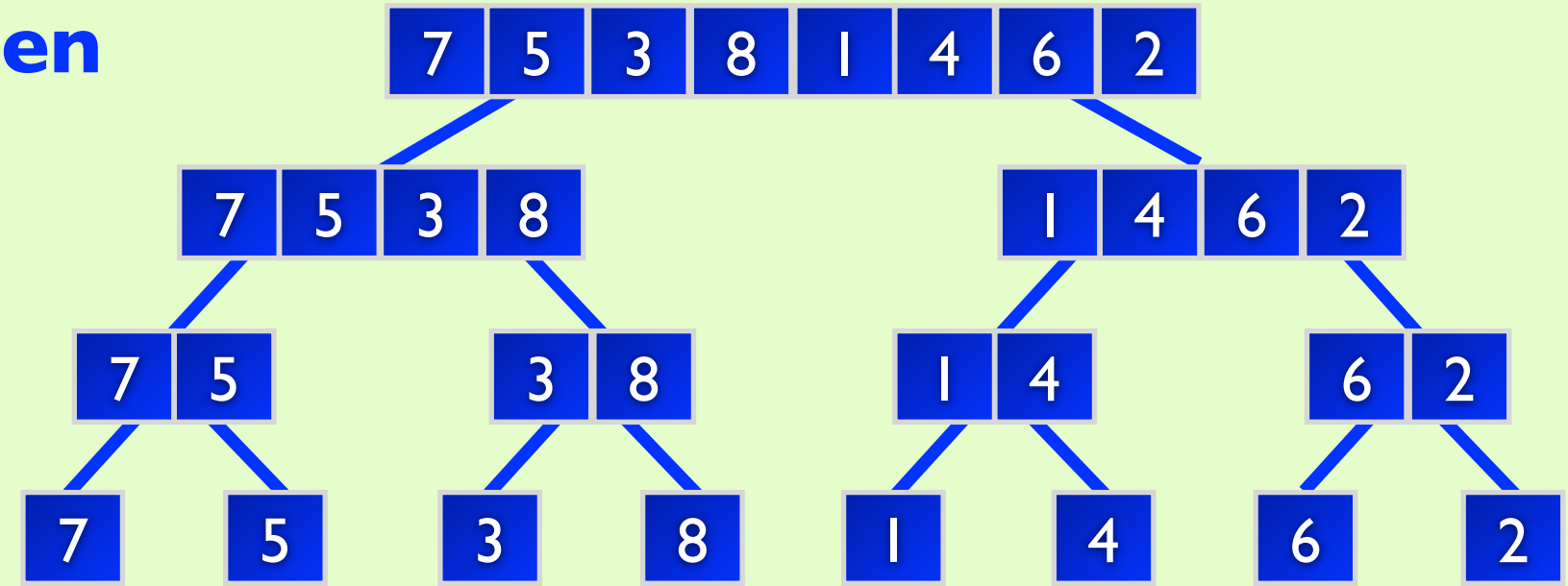


© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen

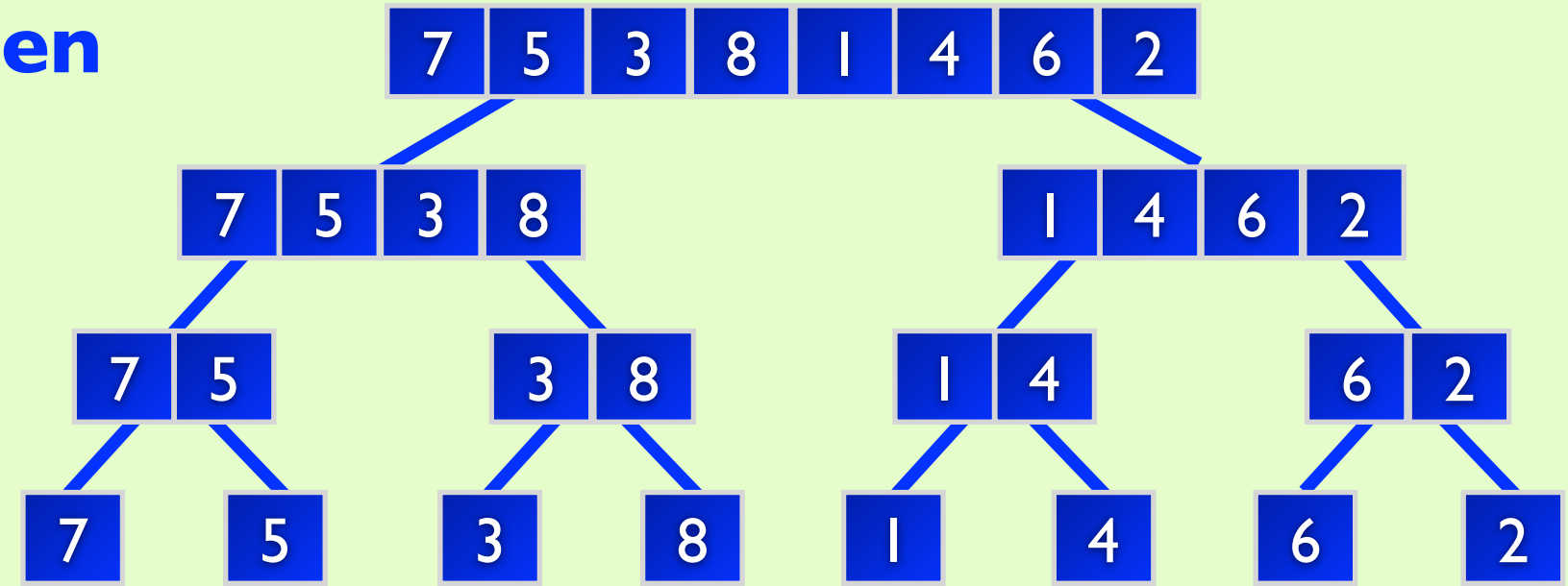


© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen



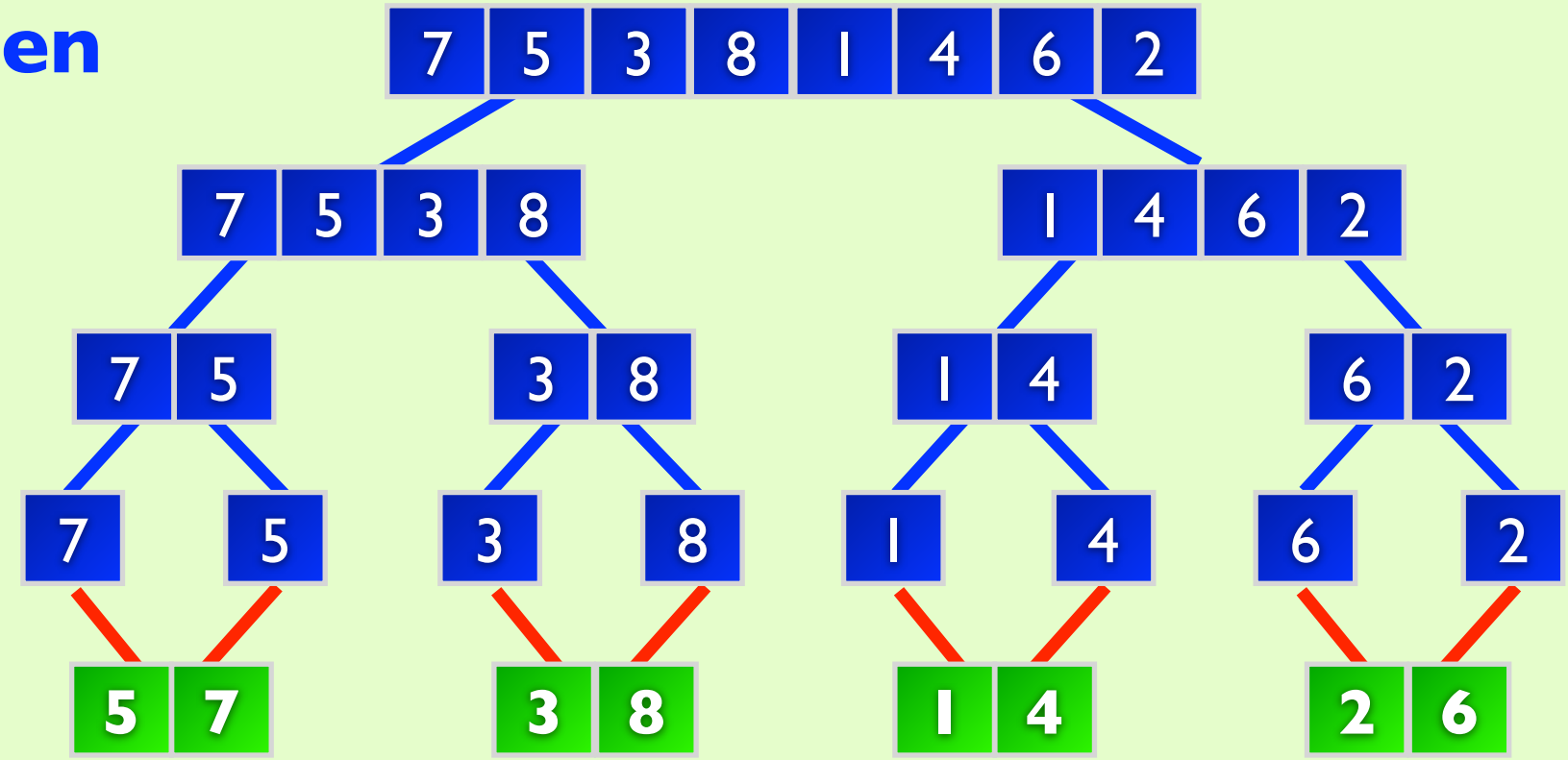
Mischen

© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen



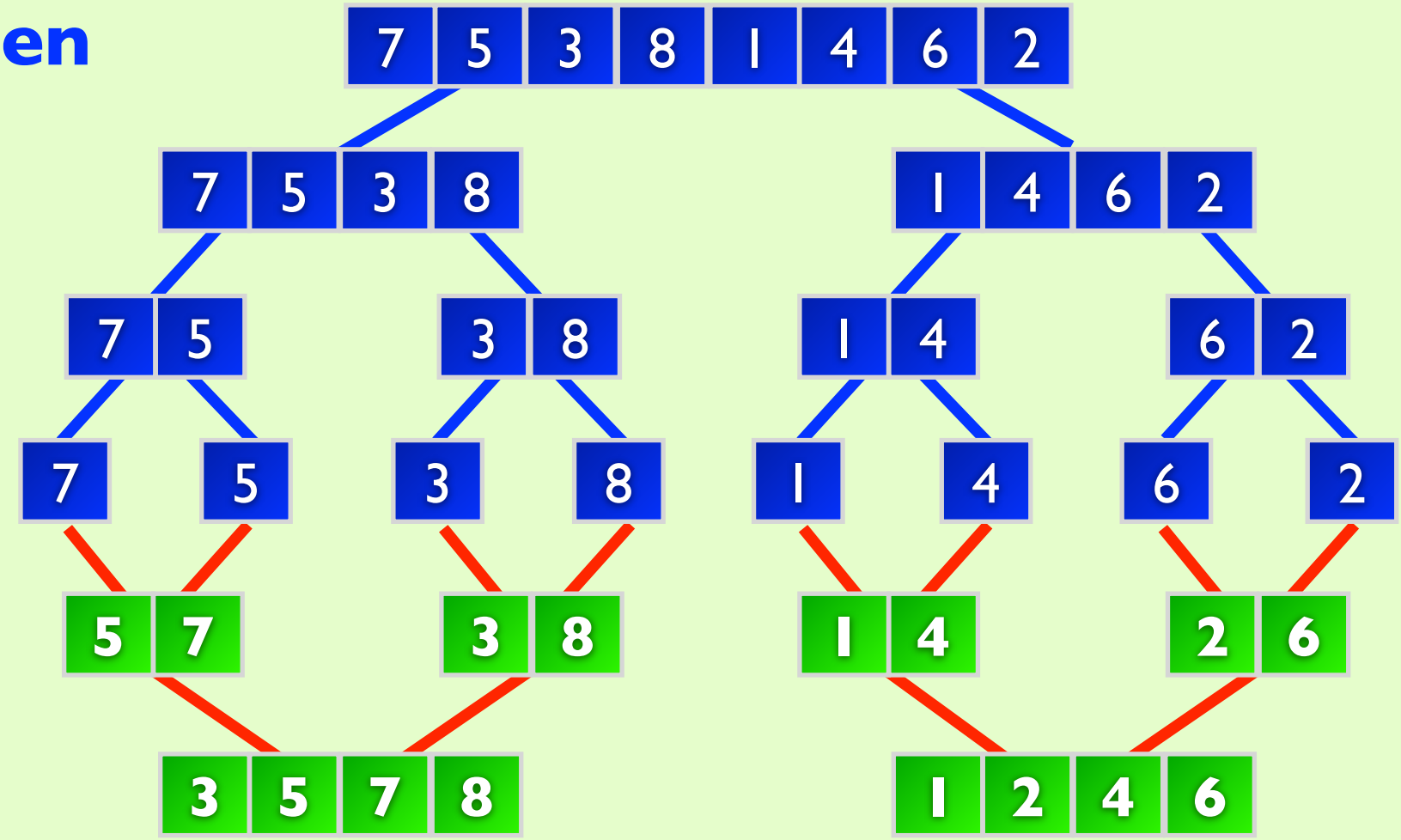
Mischen

© 2006-2023 Jörg Striegnitz

Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`

Teilen

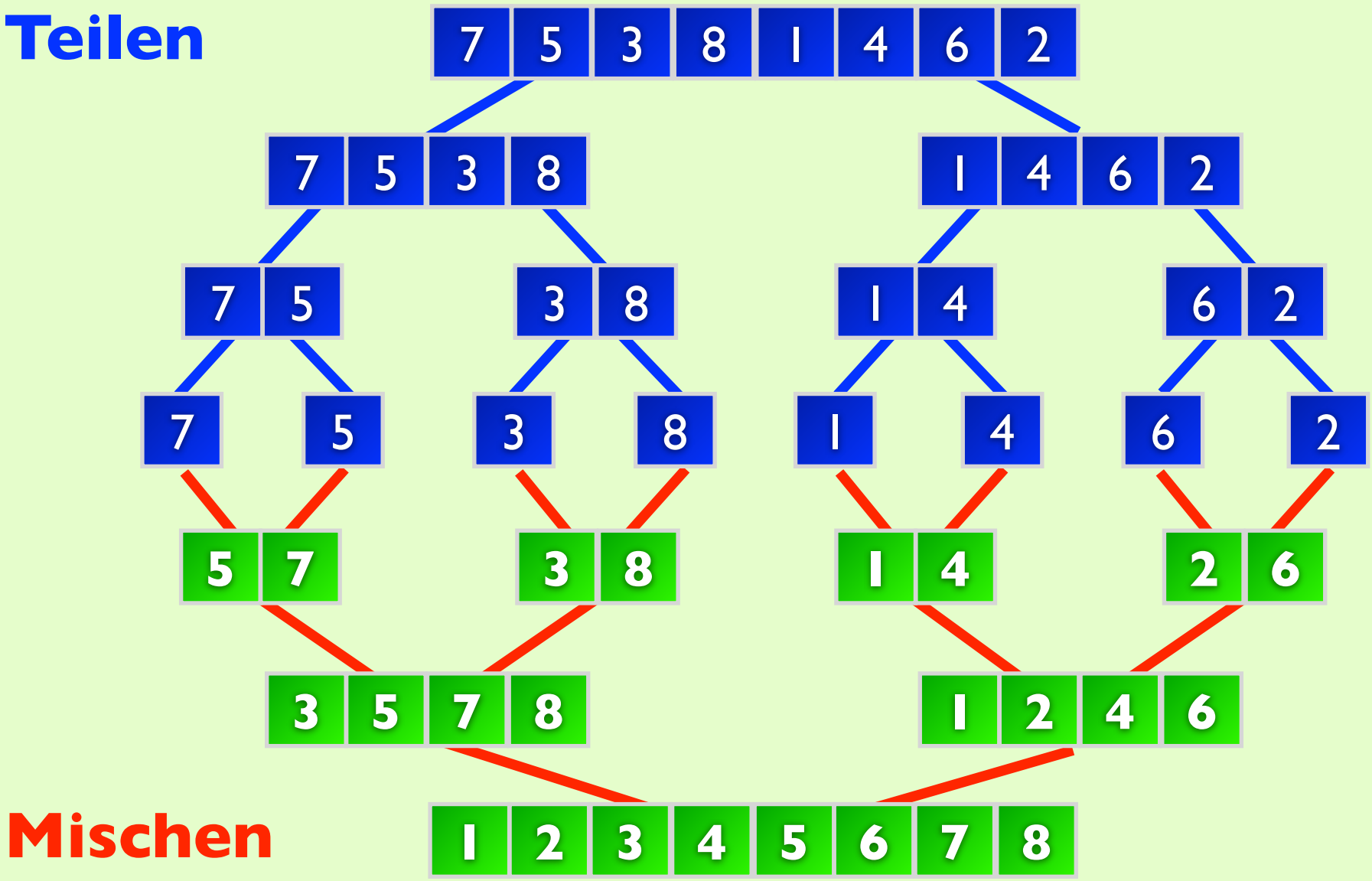


Mischen

© 2006-2023 Jörg Striegnitz

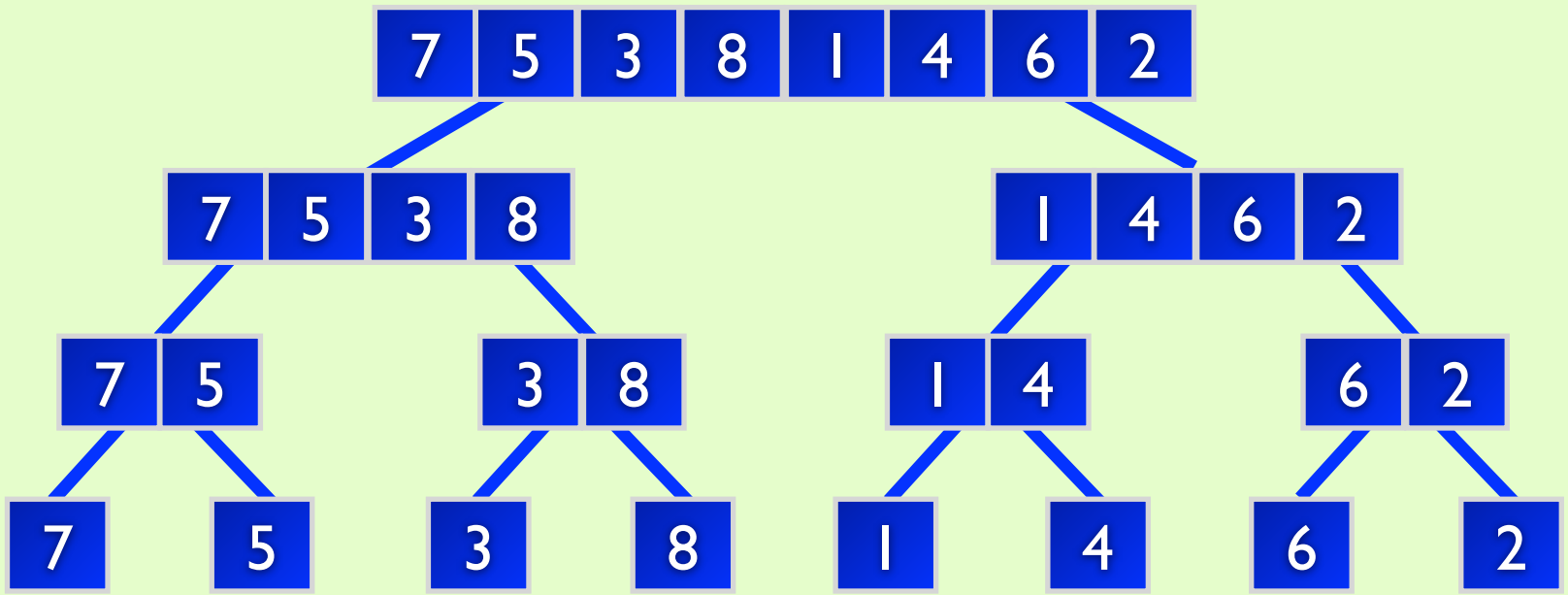
Konzeptioneller Ablauf von `MergeSort`

B Ablauf von `MergeSort`



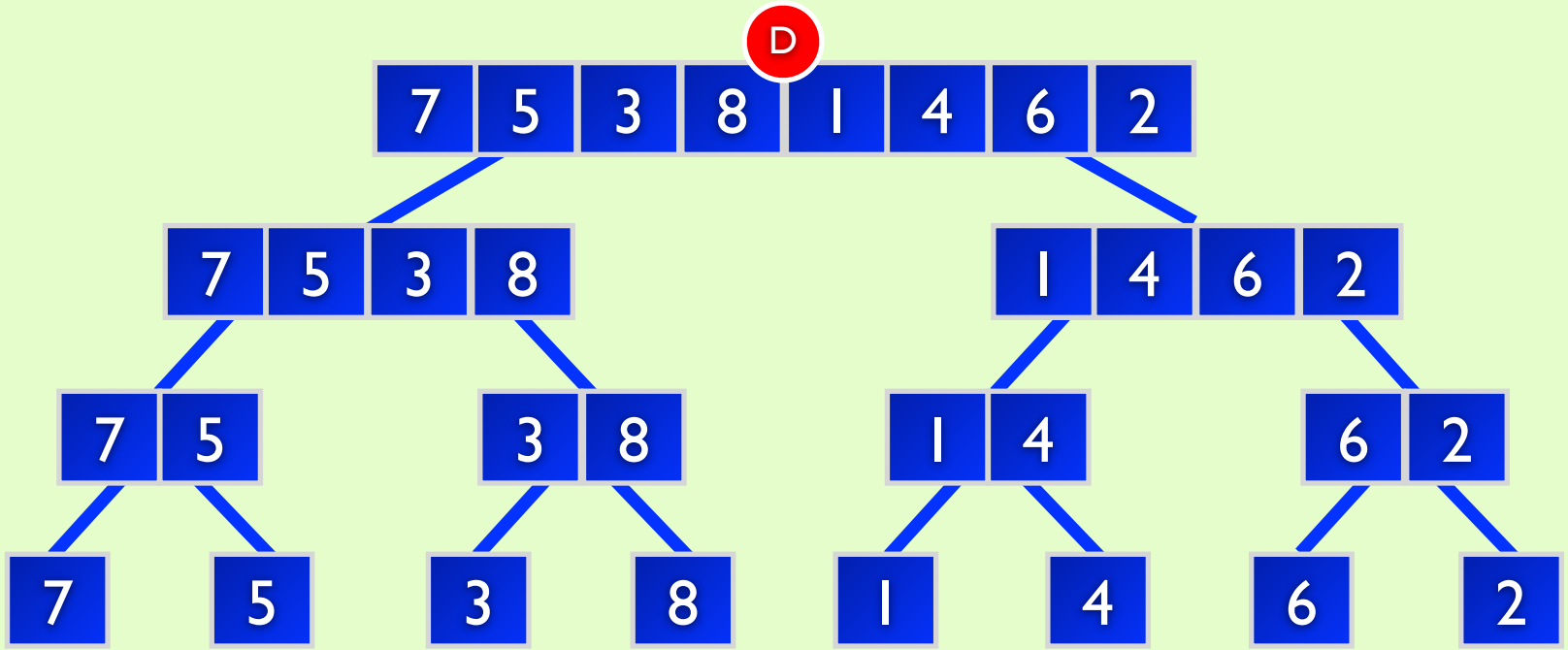
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



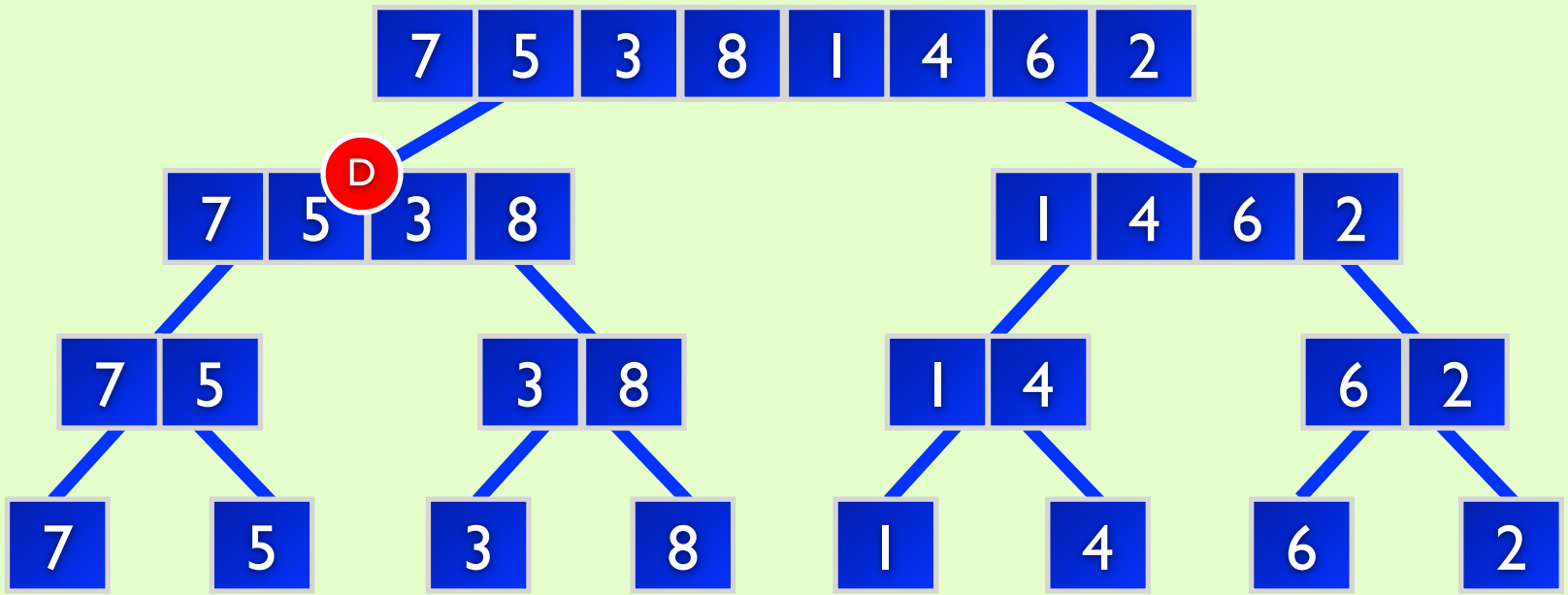
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



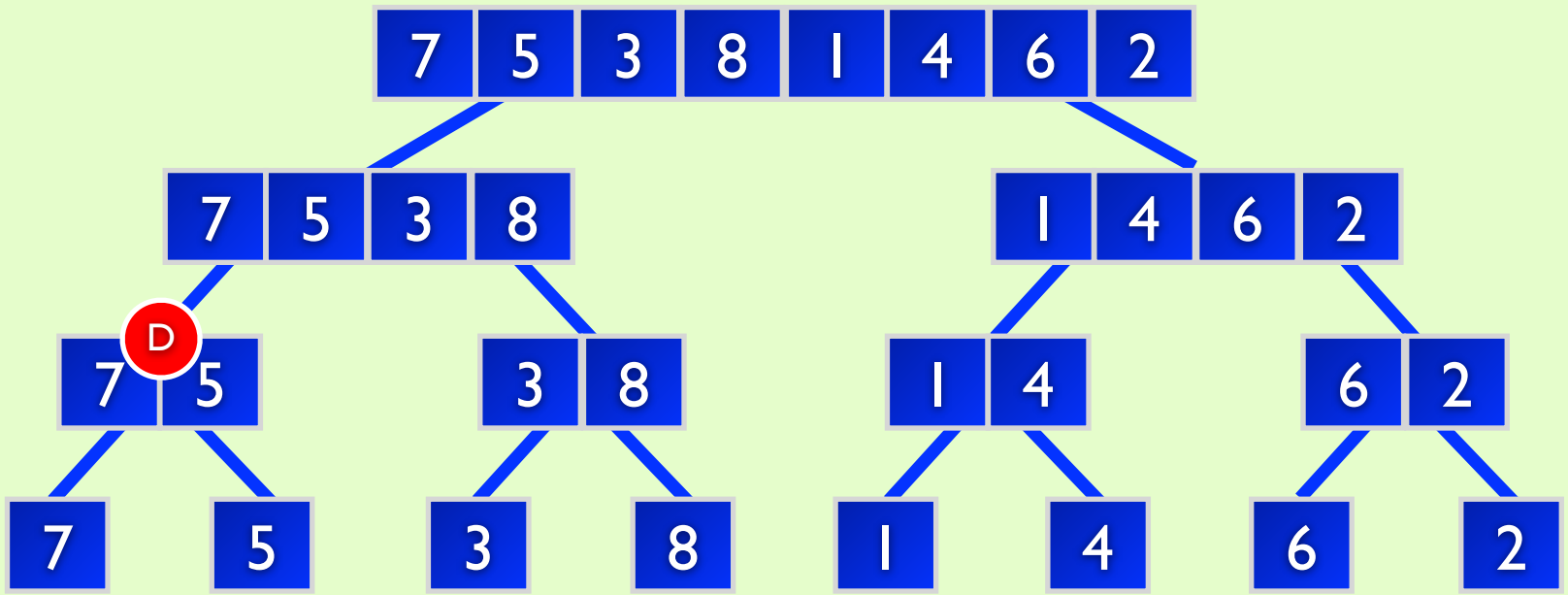
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



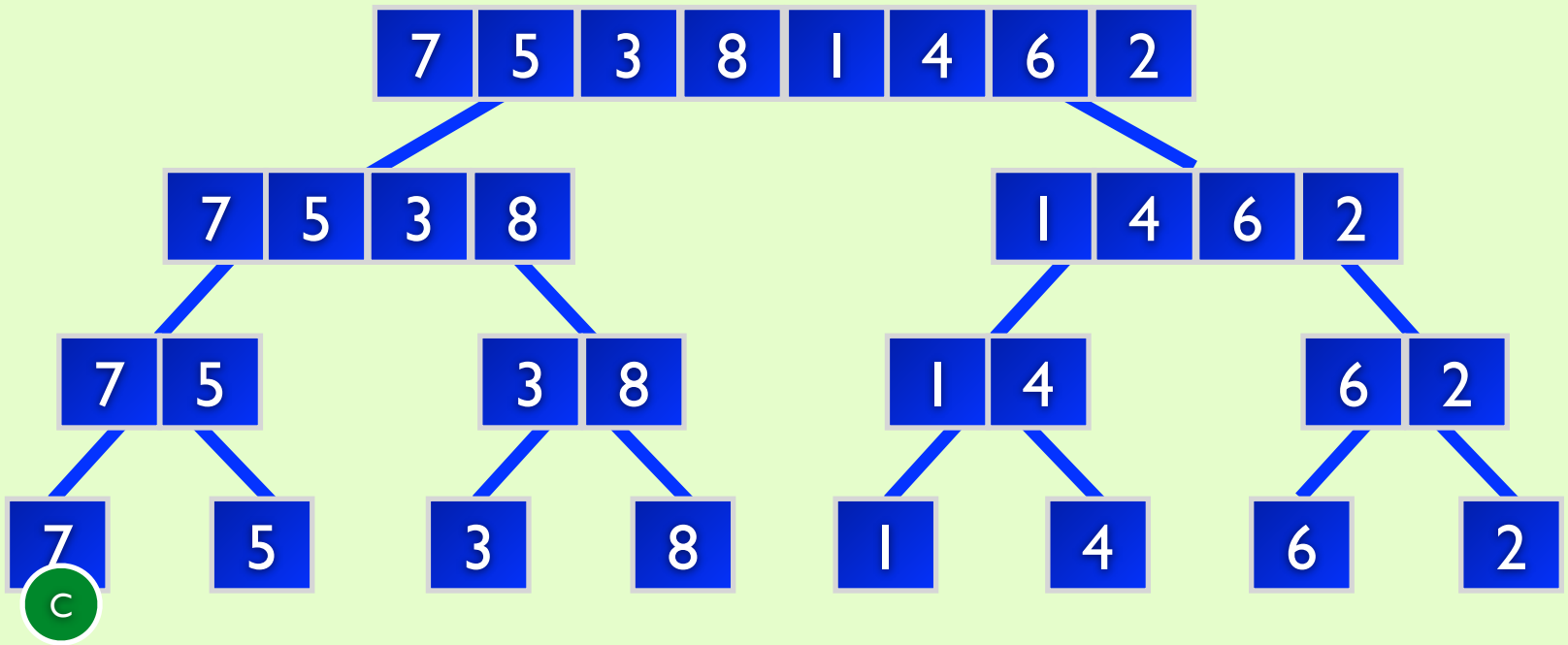
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



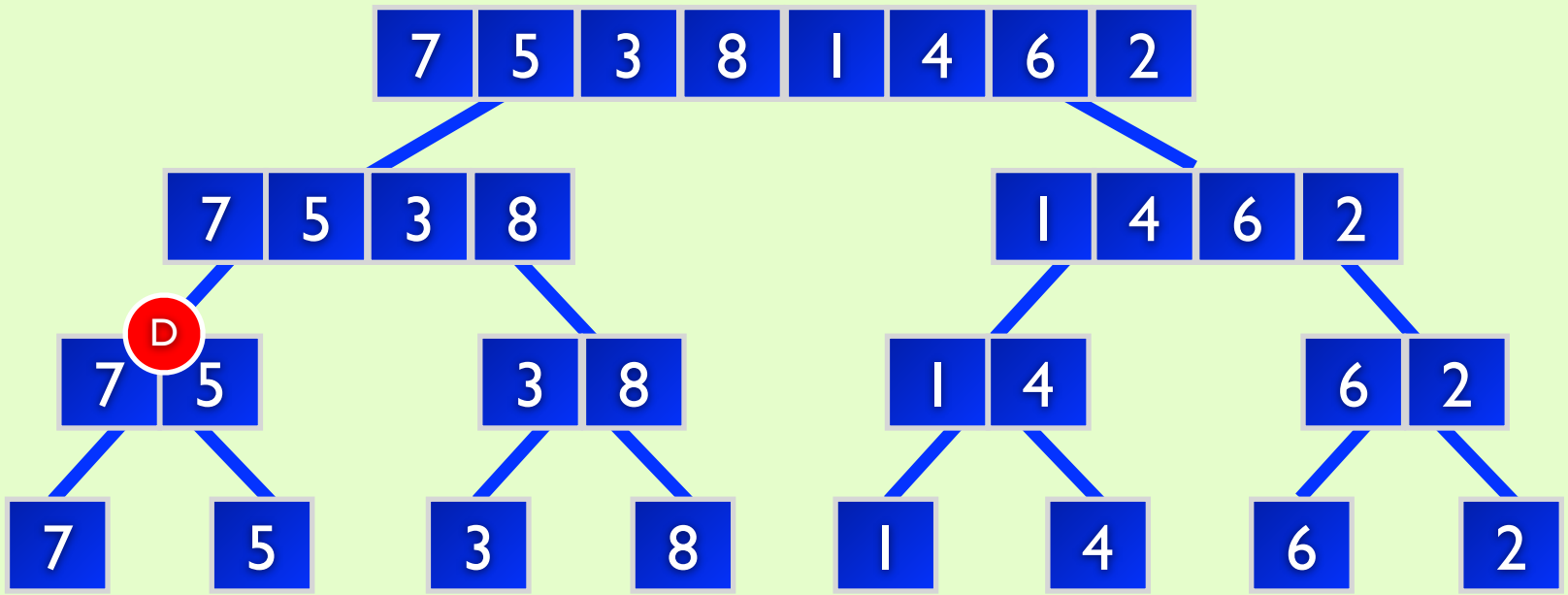
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



© 2006-2023 Jörg Striegnitz

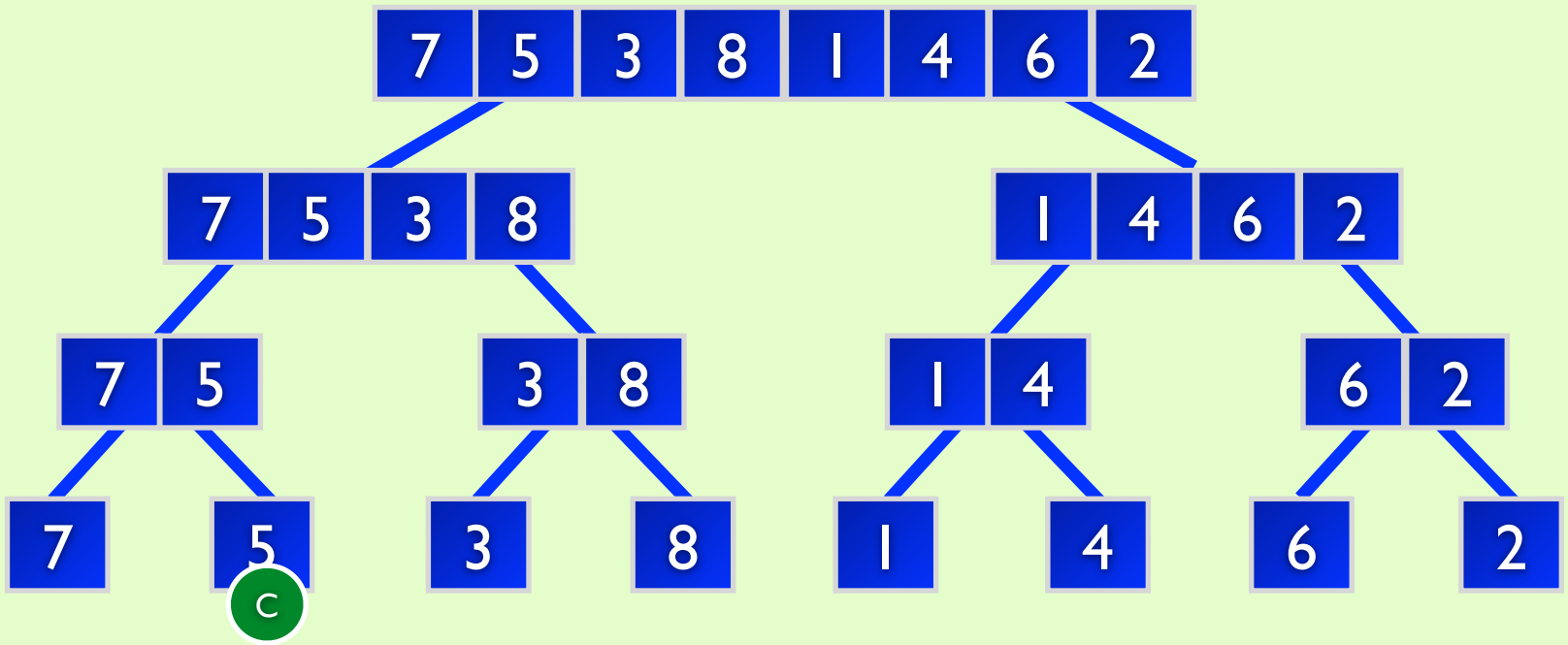
B Tatsächlicher Ablauf von MergeSort



© 2006-2023 Jörg Striegnitz

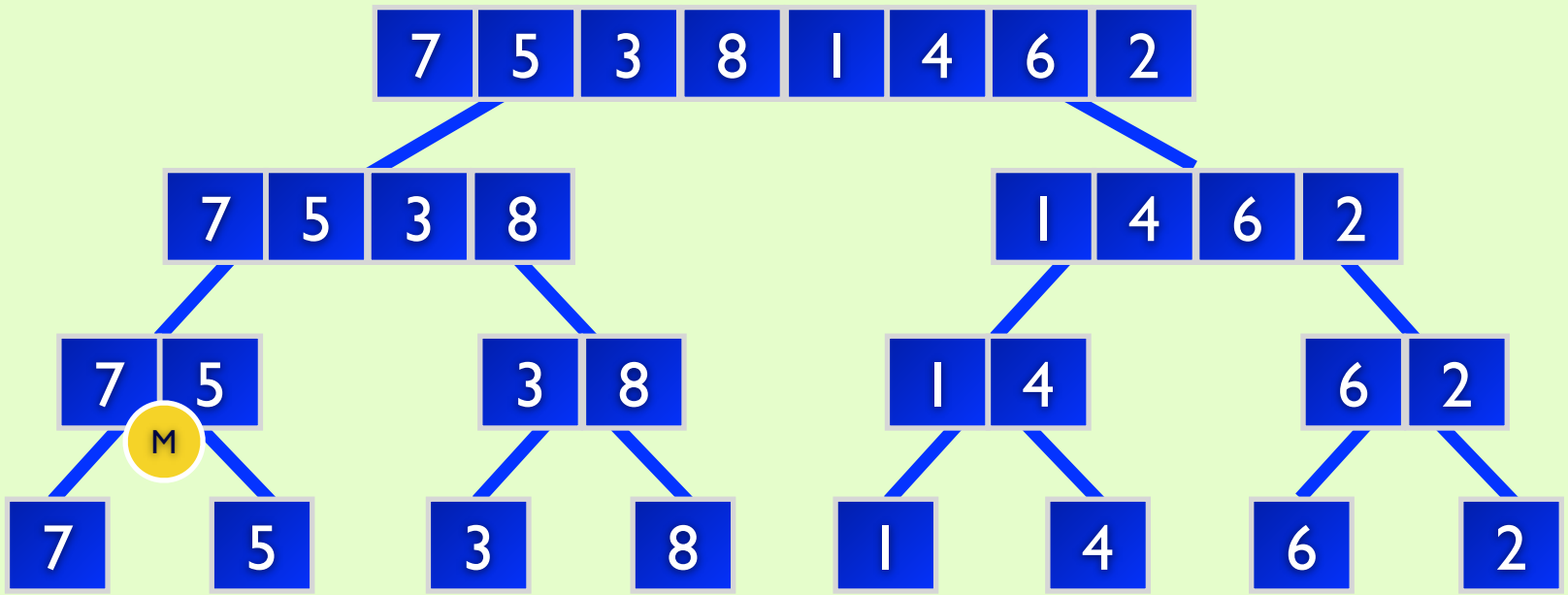
Tatsächliche Ablauf von `MergeSort`

B Tatsächlicher Ablauf von `MergeSort`



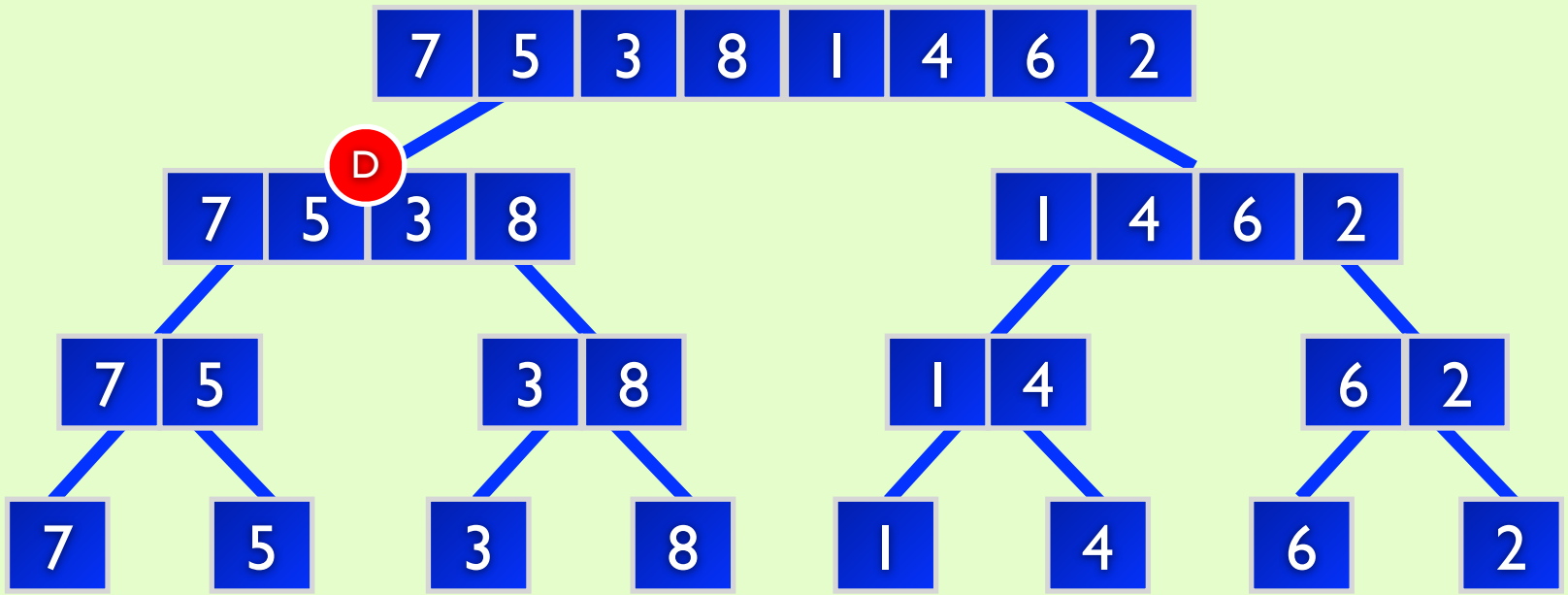
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



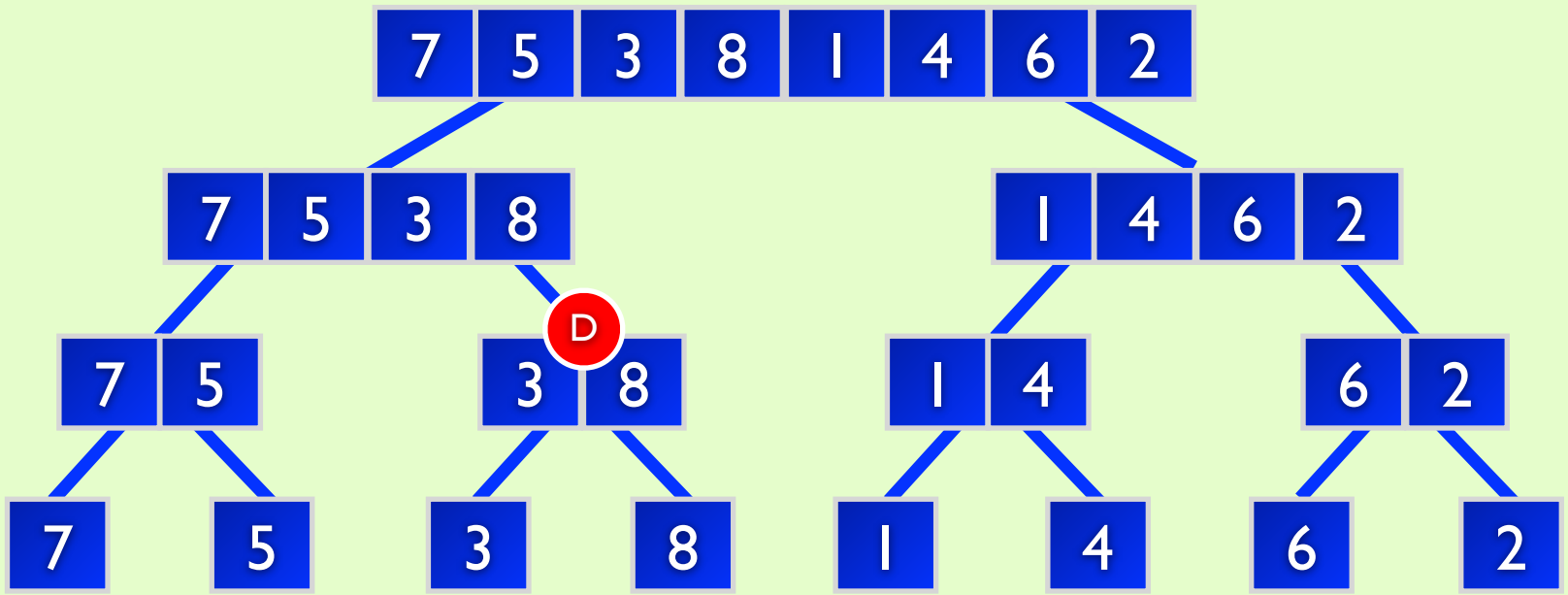
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



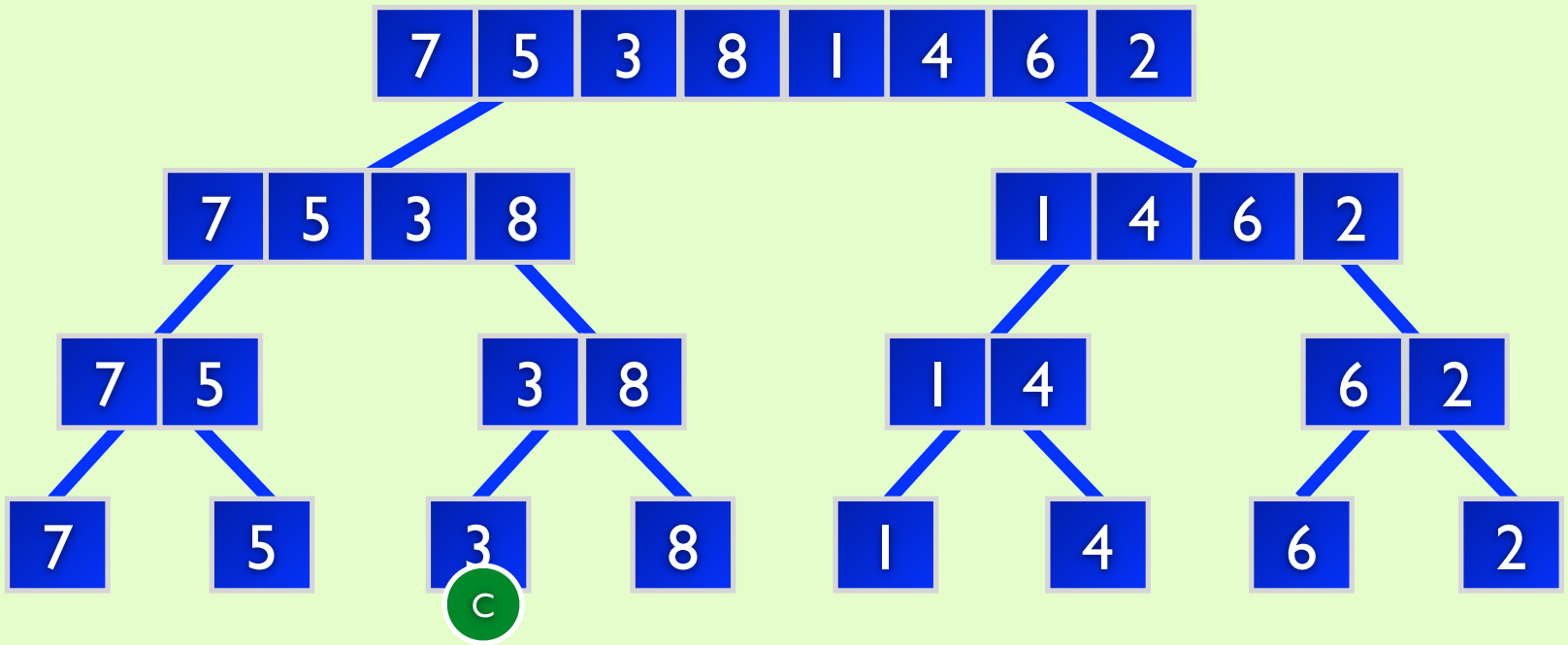
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



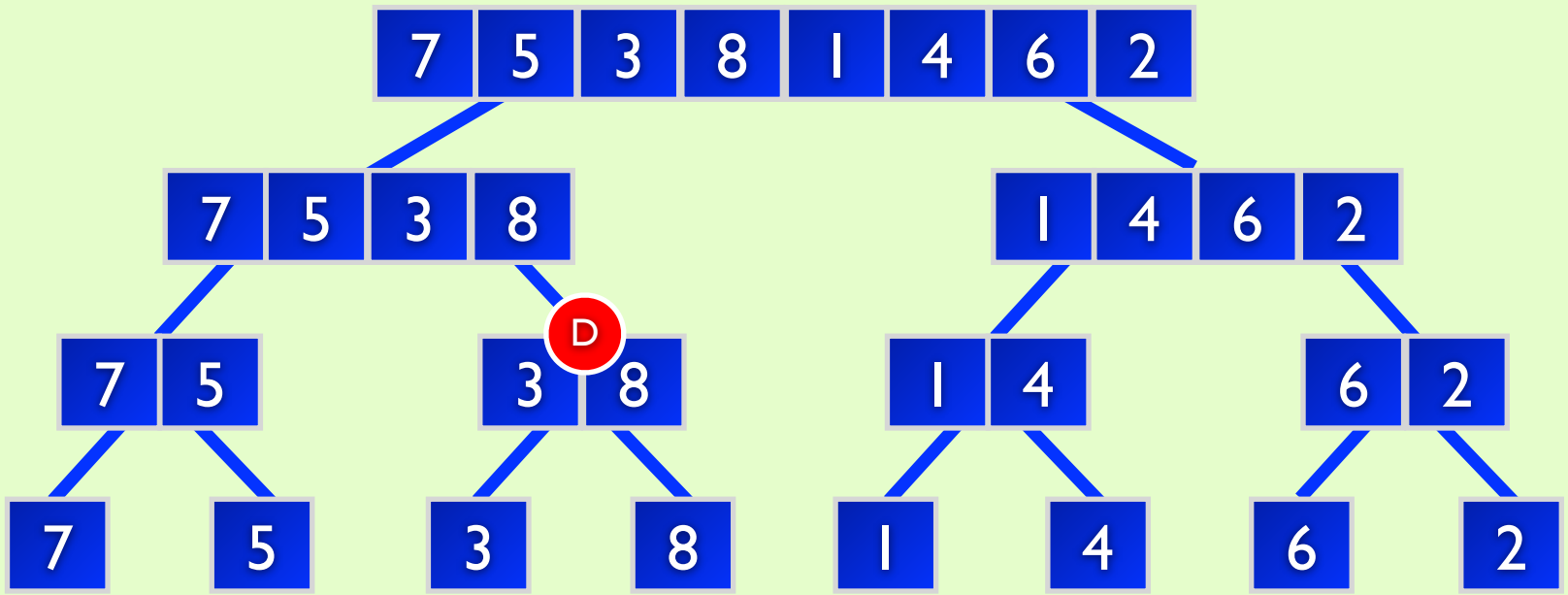
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



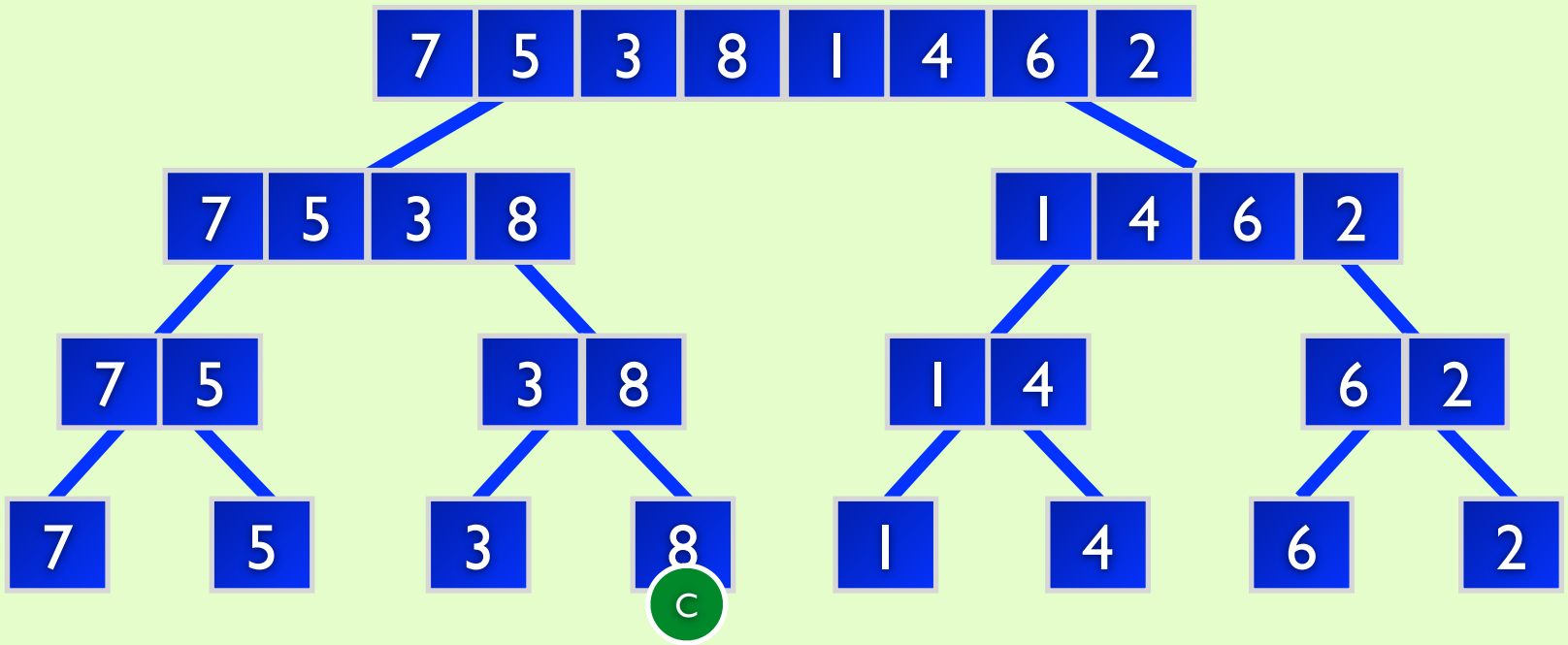
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



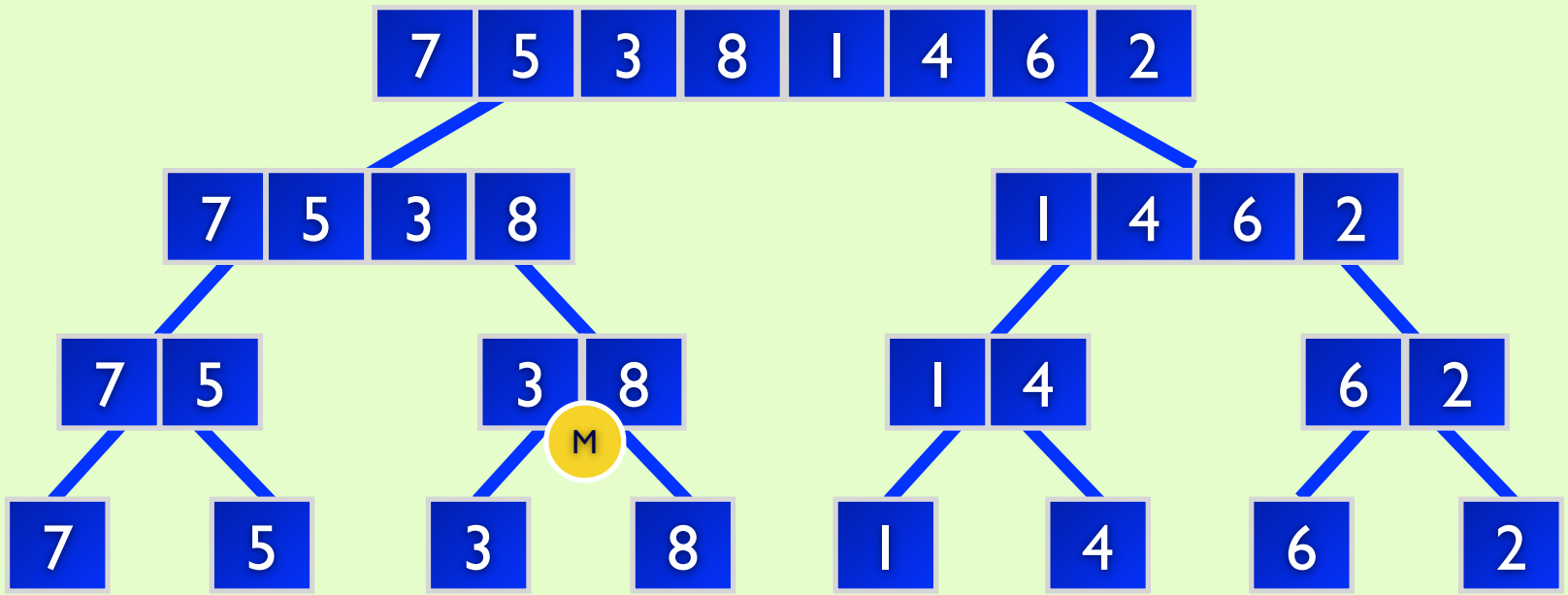
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



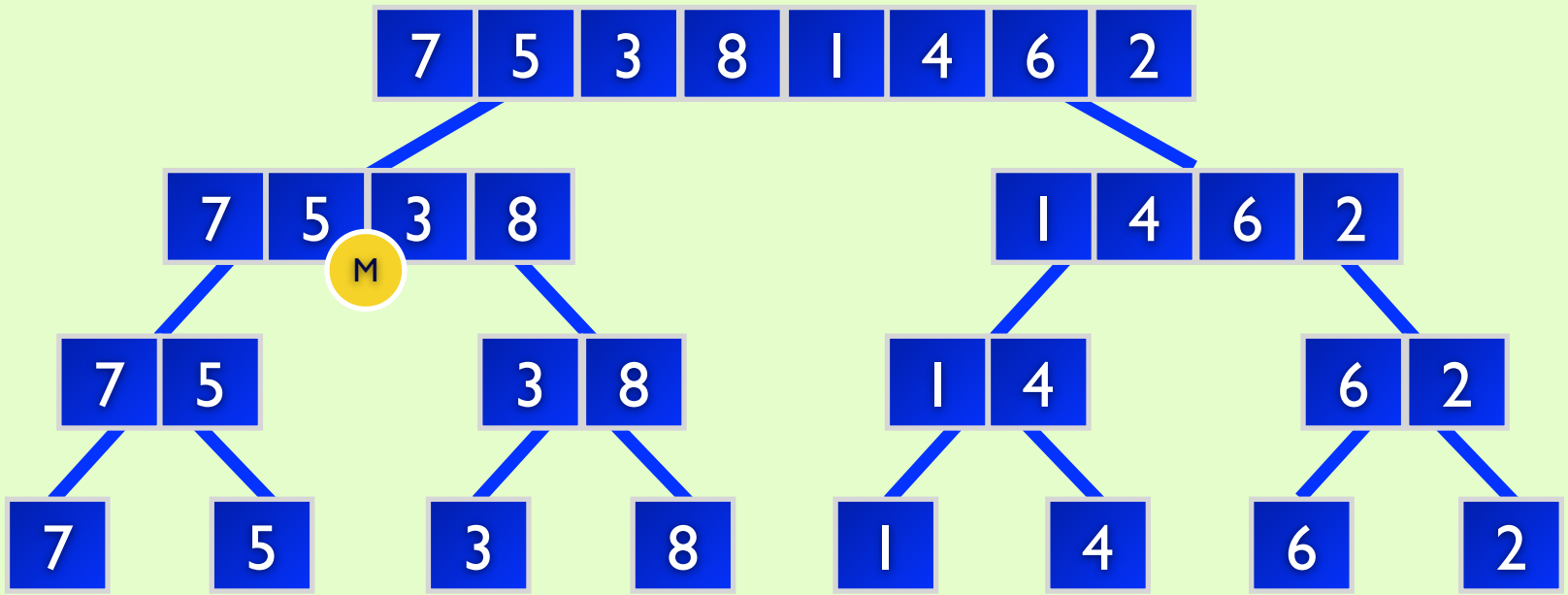
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



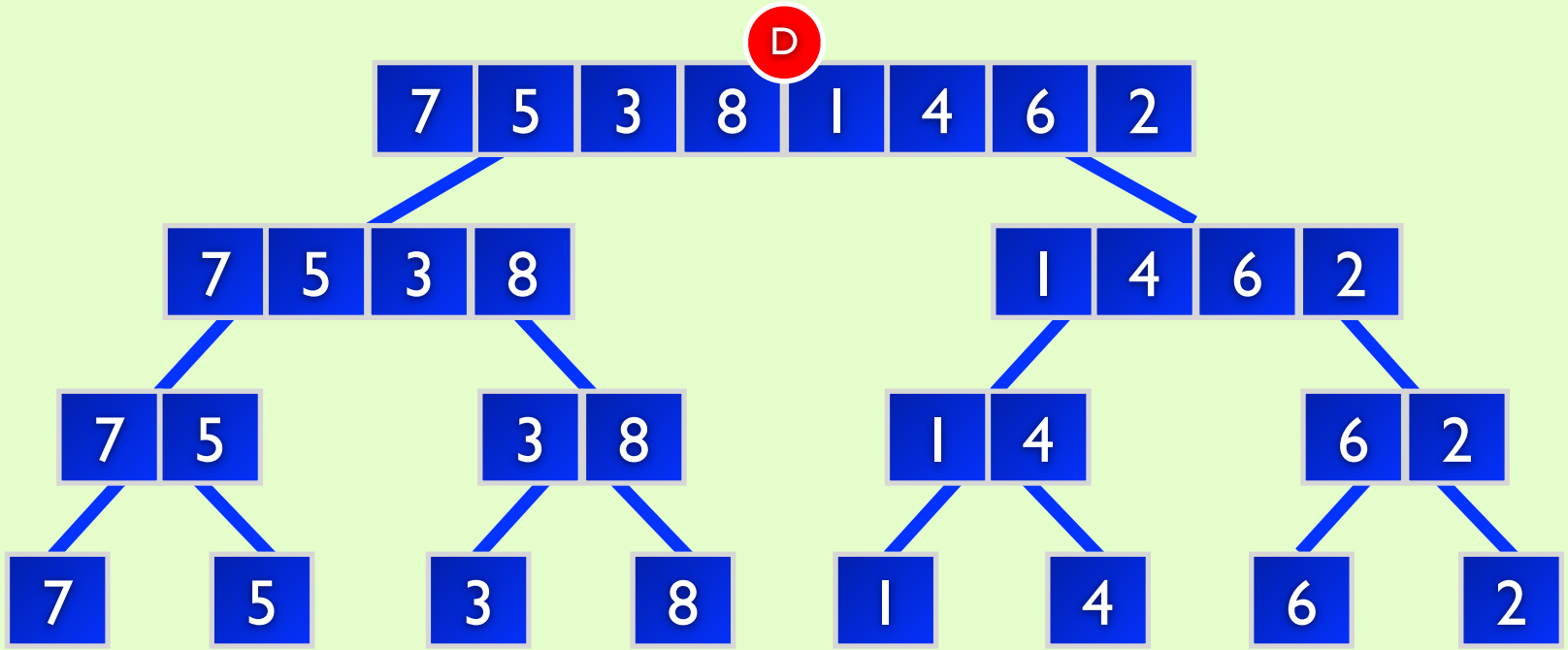
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



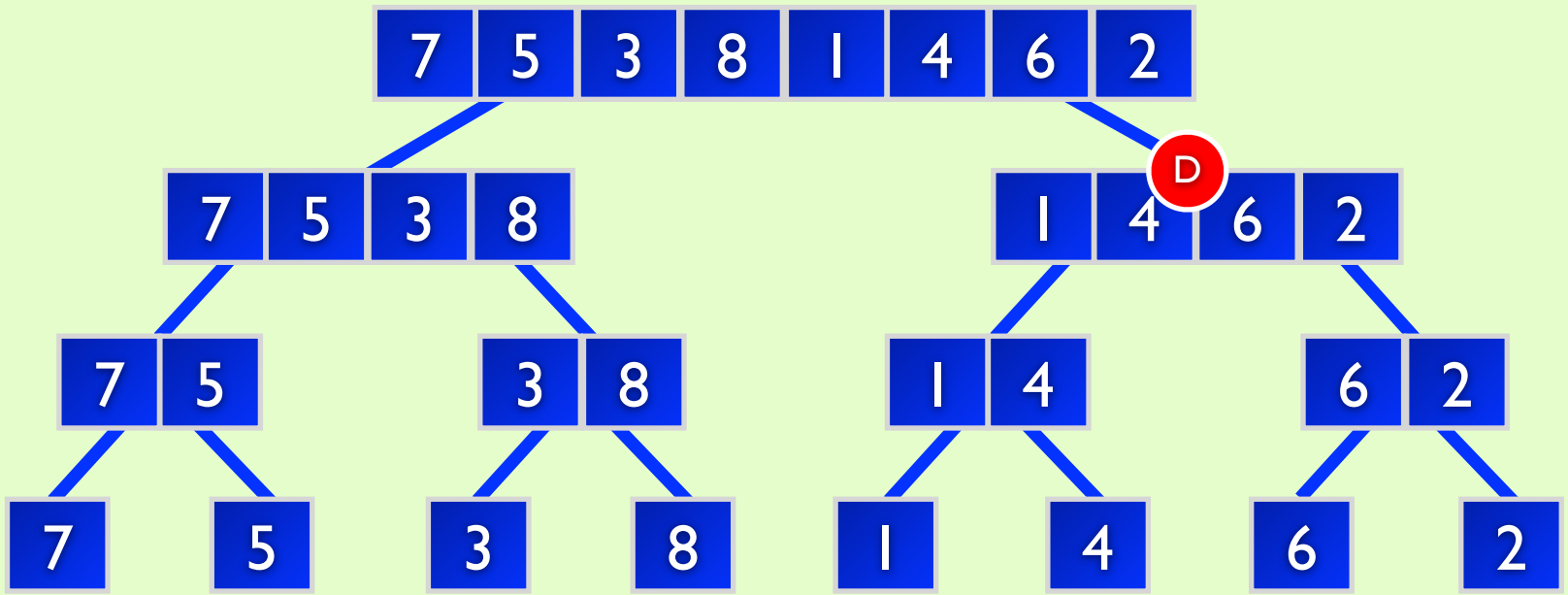
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



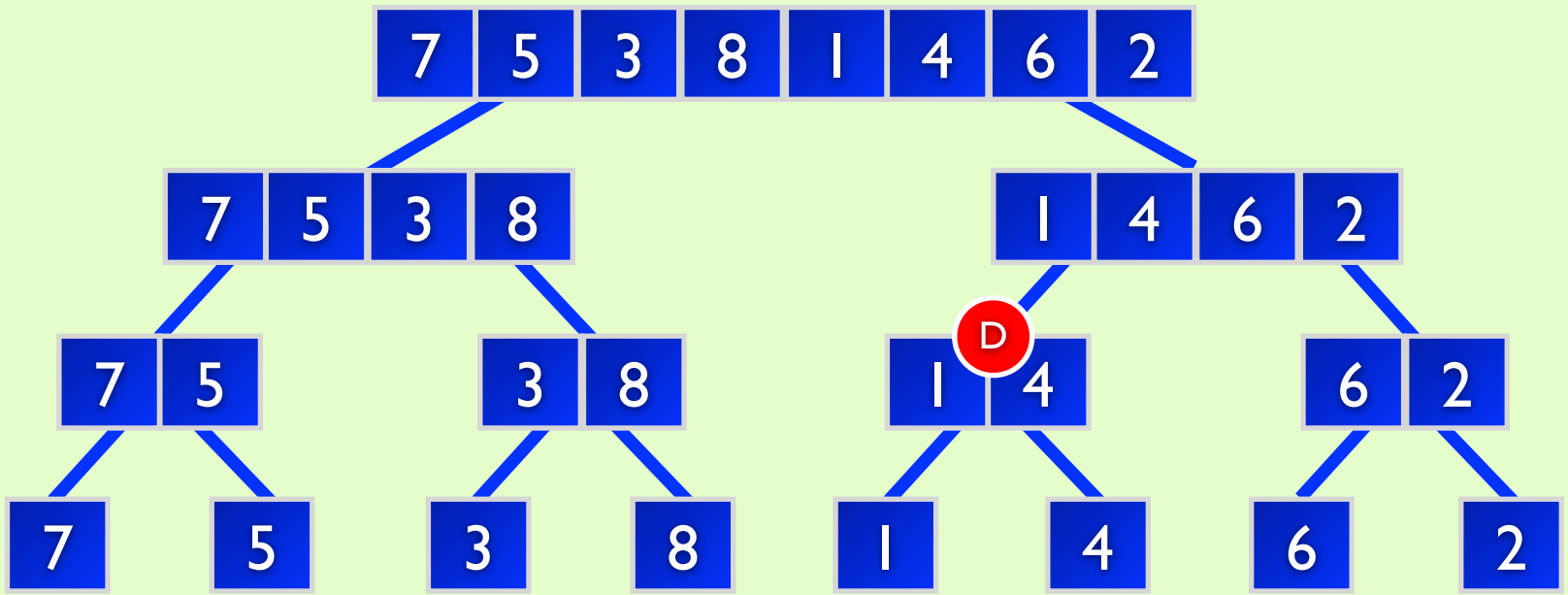
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



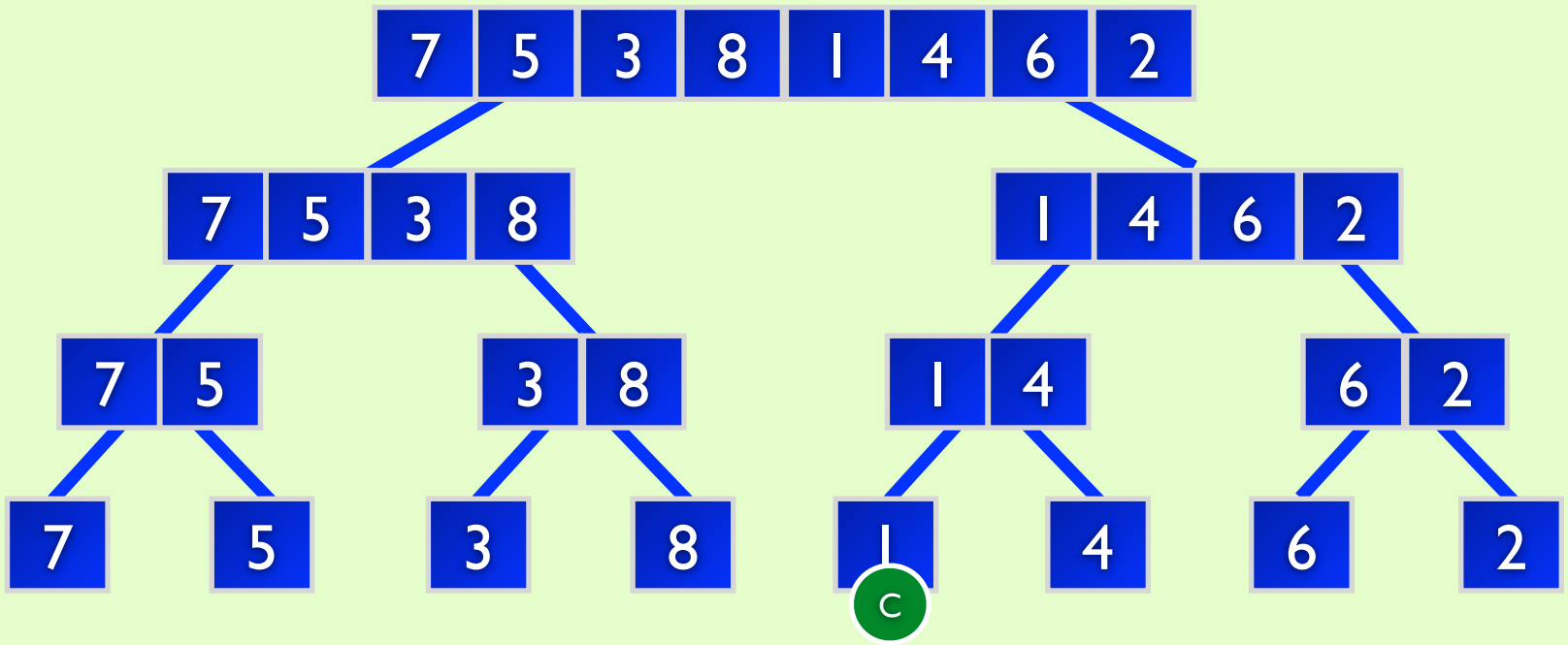
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



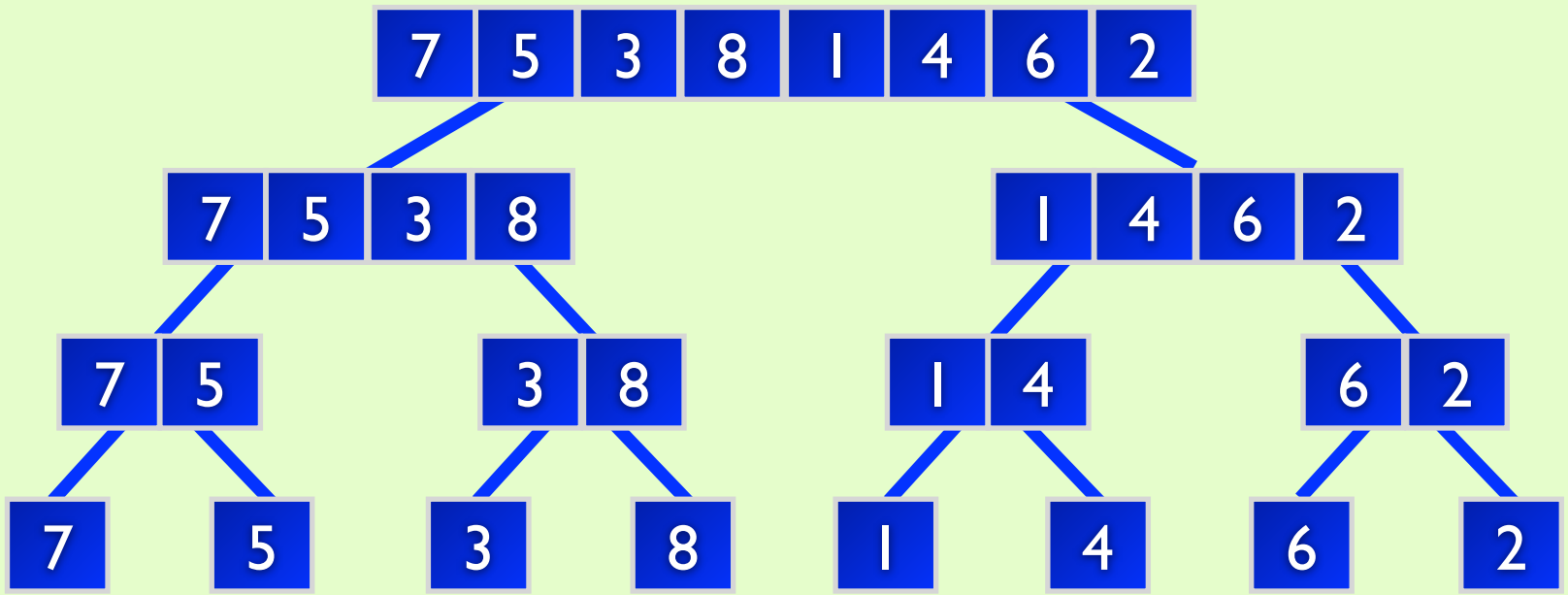
© 2006-2023 Jörg Striegnitz

B Tatsächlicher Ablauf von MergeSort



© 2006-2023 Jörg Striegnitz

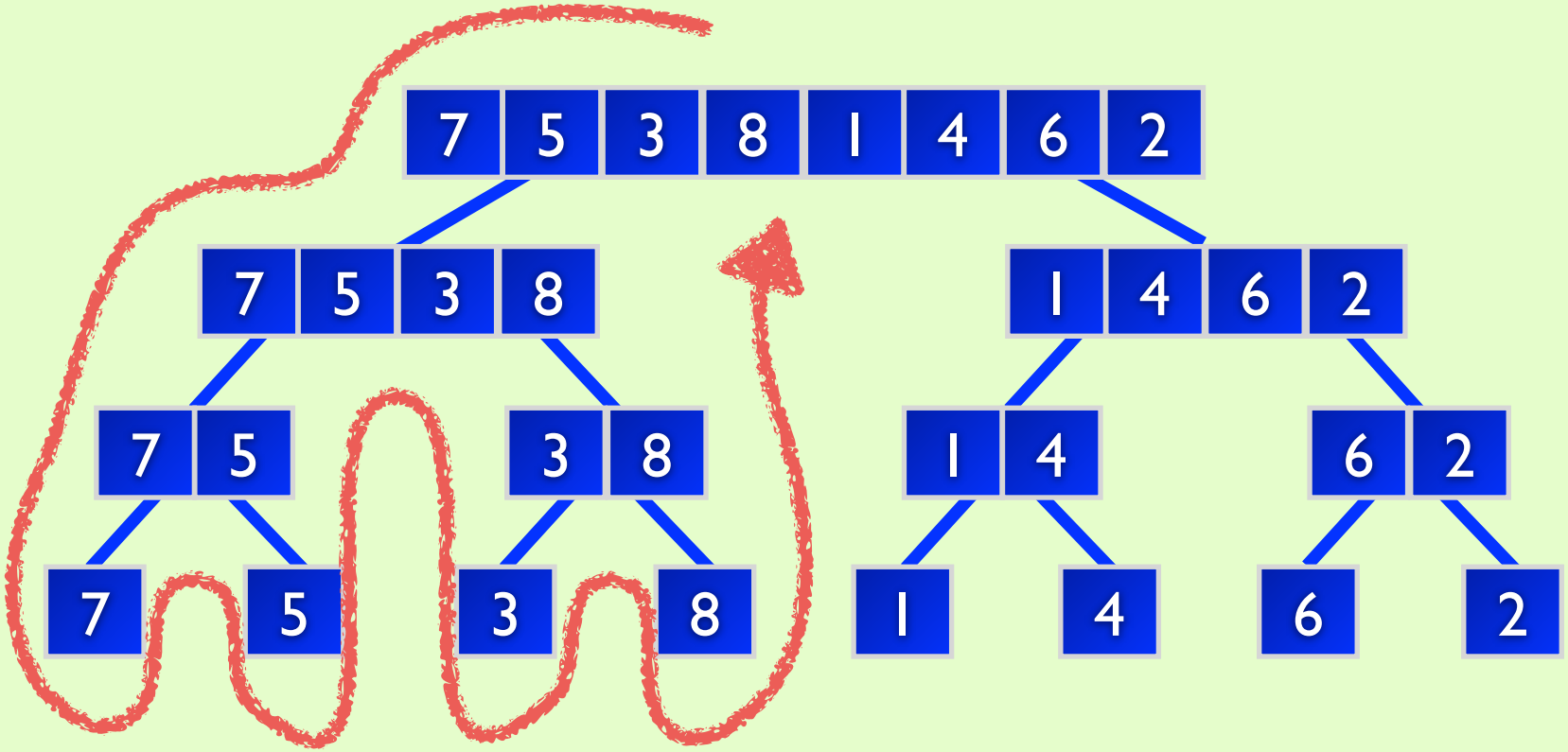
B Tatsächlicher Ablauf von MergeSort



© 2006-2023 Jörg Striegnitz

Tatsächliche Ablauf von `MergeSort`

B Tatsächlicher Ablauf von `MergeSort`

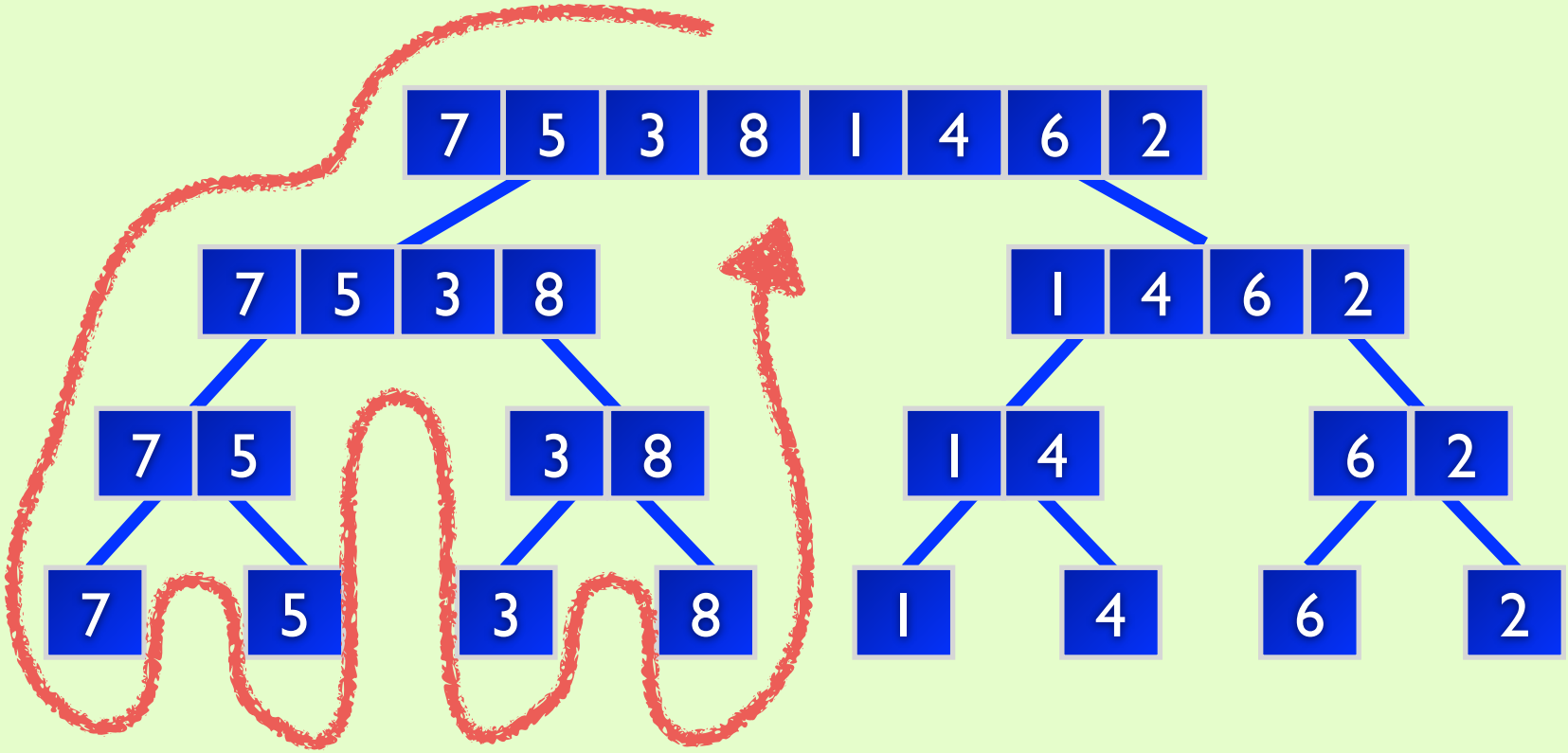


- **Ablauf entspricht „Tiefendurchlauf“**

© 2006-2023 Jörg Striegnitz

Tatsächliche Ablauf von `MergeSort`

B Tatsächlicher Ablauf von `MergeSort`



- **Ablauf entspricht „Tiefendurchlauf“**
- **gute Speicherlokalität**
  Wichtig im Zusammenhang mit Caching

© 2006-2023 Jörg Striegnitz

Merge Sort: Komplexitätsanalyse

- **Vergleiche**
  - Folge F der Länge n wird in zwei Teilfolgen der Längen $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ zerlegt.
  - Zur Verschmelzung sind $\lfloor n/2 \rfloor$ + $\lceil n/2 \rceil$ - 1 = n-1 Vergleiche erforderlich
  - Für die Gesamtzahl der Vergleiche $T(n)$ ergibt sich folgende Rekursionsgleichung:

$$T(n) = \begin{cases} 0 & \text{falls } n = 0 \vee n = 1 \\ n - 1 + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{für } n > 1 \end{cases}$$

Für diese gilt:

$$T(n) = n \cdot \lceil \mathrm{ld}\ n \rceil - 2^{\lceil \mathrm{ld}\ n \rceil} + 1$$

© 2006-2023 Jörg Striegnitz

`Merge Sort`: Abschließende Betrachtung

- **Im Vergleich zu `Quicksort` ist die Rekursionstiefe aufgrund der Teilung immer optimal!**

© 2006-2023 Jörg Striegnitz

`Merge Sort`: Abschließende Betrachtung

- **Im Vergleich zu `Quicksort` ist die Rekursionstiefe aufgrund der Teilung immer optimal!**

- **`Merge Sort` ist stabil aber kein *in-situ*-Verfahren**
  - Hilfsfeld beim Mischen
  - *in-situ*-Verfahren ist möglich, erfordert aber erheblich mehr Speicherbewegungen

© 2006-2023 Jörg Striegnitz

`Merge Sort`: Abschließende Betrachtung

- **Im Vergleich zu `Quicksort` ist die Rekursionstiefe aufgrund der Teilung immer optimal!**
- **`Merge Sort` ist stabil aber kein *in-situ*-Verfahren**
  - Hilfsfeld beim Mischen
  - *in-situ*-Verfahren ist möglich, erfordert aber erheblich mehr Speicherbewegungen
- **Nicht-rekursive Variante - „*Bottom-up* `Merge Sort`“**
  (Ohne Stack, da im Gegensatz zu `Quicksort` vor dem rekursiven Aufruf nichts verändert wird)

© 2006-2023 Jörg Striegnitz

`Merge Sort`: Abschließende Betrachtung

- **Im Vergleich zu `Quicksort` ist die Rekursionstiefe aufgrund der Teilung immer optimal!**
- **`Merge Sort` ist stabil aber kein *in-situ*-Verfahren**
  - Hilfsfeld beim Mischen
  - *in-situ*-Verfahren ist möglich, erfordert aber erheblich mehr Speicherbewegungen
- **Nicht-rekursive Variante - „*Bottom-up* `Merge Sort`“**
  (Ohne Stack, da im Gegensatz zu `Quicksort` vor dem rekursiven Aufruf nichts verändert wird)
- **`MergeSort` eignet sich gut für das Sortieren von Daten auf Hintergrundspeichern** (z.B. Festplatten / Bänder)
  - **Grund**: Daten werden nur sequentiell durchlaufen (Mischen)

© 2006-2023 Jörg Striegnitz

`Merge Sort`: Abschließende Betrachtung

- **Im Vergleich zu `Quicksort` ist die Rekursionstiefe aufgrund der Teilung immer optimal!**
- **`Merge Sort` ist stabil aber kein *in-situ*-Verfahren**
  - Hilfsfeld beim Mischen
  - *in-situ*-Verfahren ist möglich, erfordert aber erheblich mehr Speicherbewegungen
- **Nicht-rekursive Variante - „*Bottom-up* `Merge Sort`“**
  (Ohne Stack, da im Gegensatz zu `Quicksort` vor dem rekursiven Aufruf nichts verändert wird)
- **`MergeSort` eignet sich gut für das Sortieren von Daten auf Hintergrundspeichern** (z.B. Festplatten / Bänder)
  - **Grund**: Daten werden nur sequentiell durchlaufen (Mischen)
- **Optimierungen** - diverse denkbar; z.B.:
  - **Conquer**: kleine Felder z.B. mit `Insertion Sort` sortieren
  - Kleinstes Element aus F_1 größer als größtes aus F_2 - dann einfach aneinanderhängen
  - Globaler „Merge-Buffer“ vermeidet wiederkehrende Allokationen
  - Weitere Ausnutzung der Vorsortierung (natürlicher `Merge Sort` - s.n.F.)

© 2006-2023 Jörg Striegnitz

Idee: Natürlicher `Merge Sort`

B Natürlicher `MergeSort` I

- **Ausgangsfolge**

3	9	8	4	6	8	2	1	7

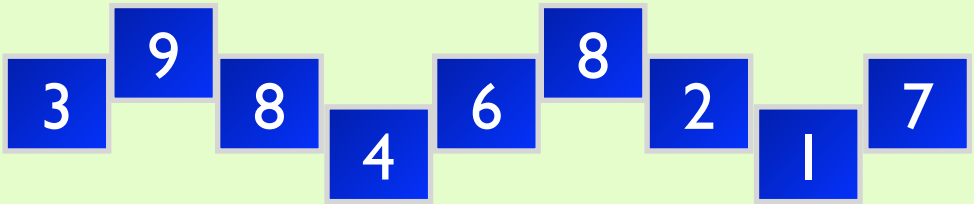
© 2006-2023 Jörg Striegnitz

Idee: Natürlicher `Merge Sort`

B Natürlicher `MergeSort` I

- **Ausgangsfolge**

- **Auf- und absteigende Sequenzen**



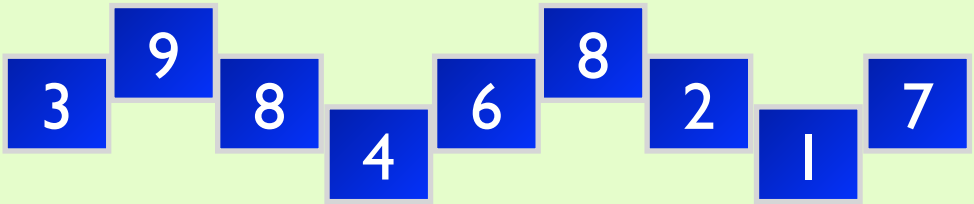
© 2006-2023 Jörg Striegnitz

Idee: Natürlicher `Merge Sort`

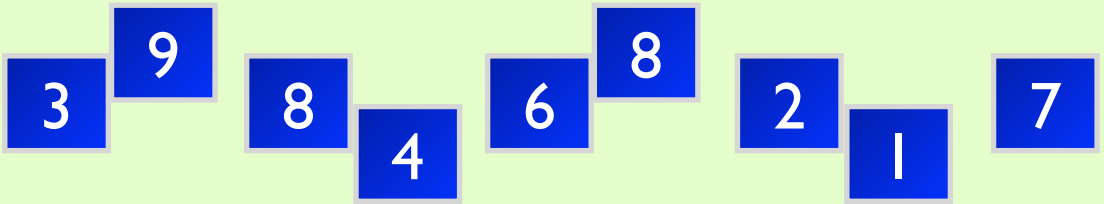
B Natürlicher `MergeSort` I

- **Ausgangsfolge**

- **Auf- und absteigende Sequenzen**



- **Teilen**



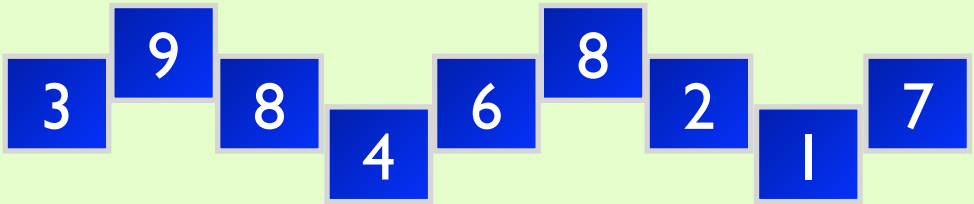
© 2006-2023 Jörg Striegnitz

Idee: Natürlicher `Merge Sort`

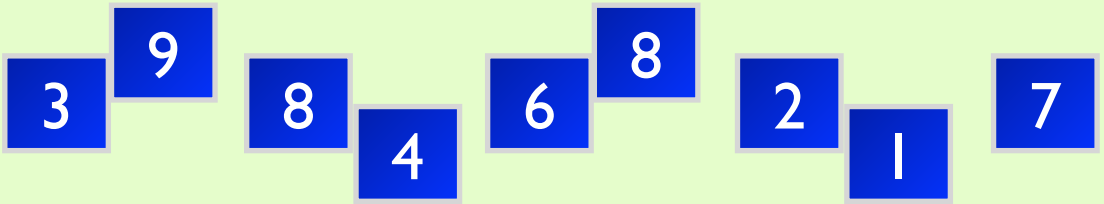
B Natürlicher `MergeSort` I

- **Ausgangsfolge**

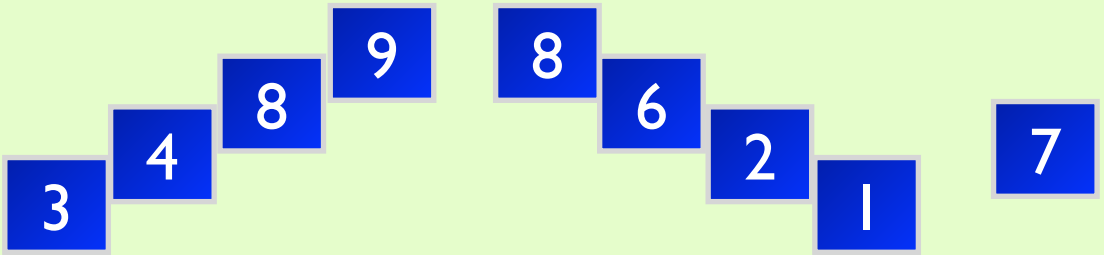
- **Auf- und absteigende Sequenzen**



- **Teilen**



- **Zusammenfassen von wechselnden Folgen**

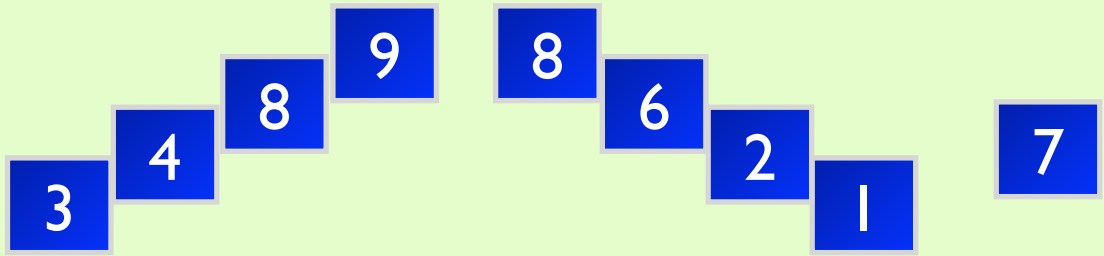


© 2006-2023 Jörg Striegnitz

Idee: Natürlicher `Merge Sort`

B Natürlicher `MergeSort` II

- **Zusammenfassen von wechselnden Folgen**

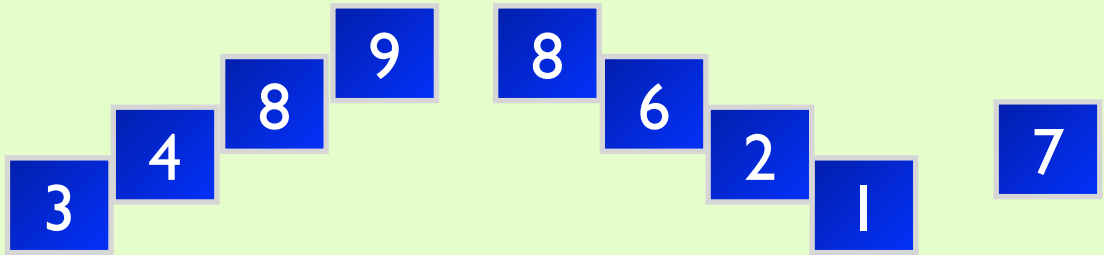


© 2006-2023 Jörg Striegnitz

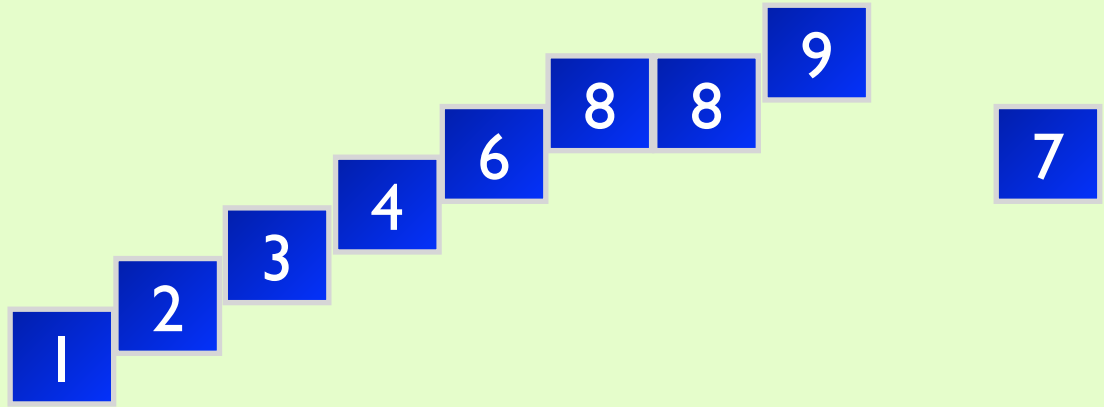
Idee: Natürlicher `Merge Sort`

B Natürlicher `MergeSort` II

- **Zusammenfassen von wechselnden Folgen**



- **Zusammenfassen von wechselnden Folgen**

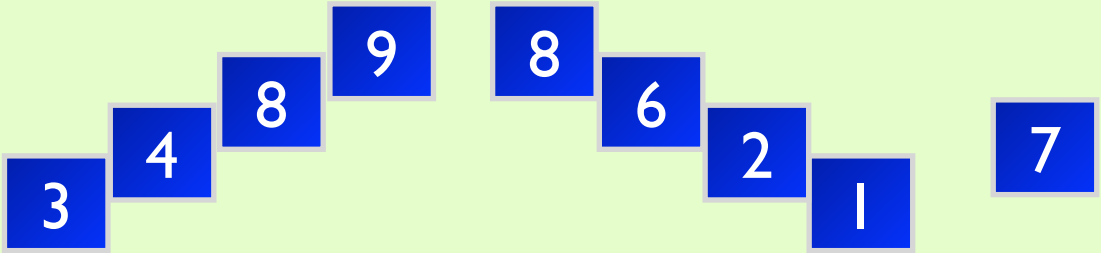


© 2006-2023 Jörg Striegnitz

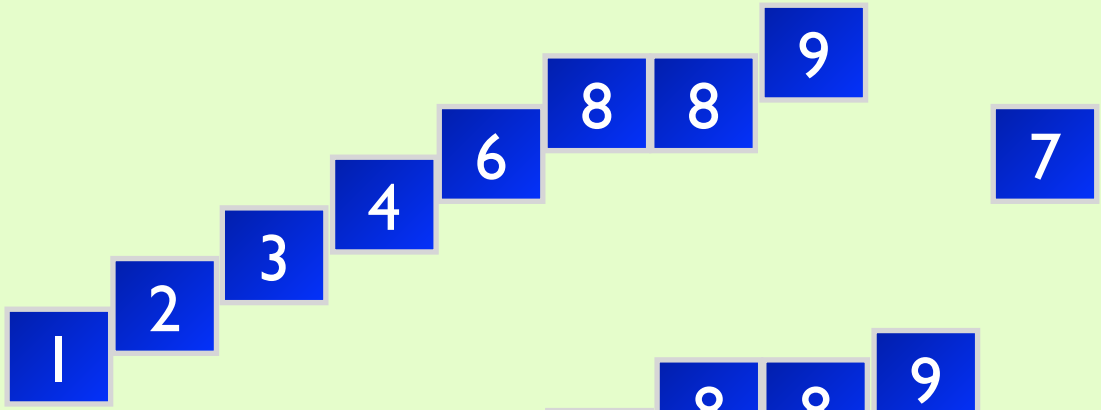
Idee: Natürlicher `Merge Sort`

B Natürlicher `MergeSort` II

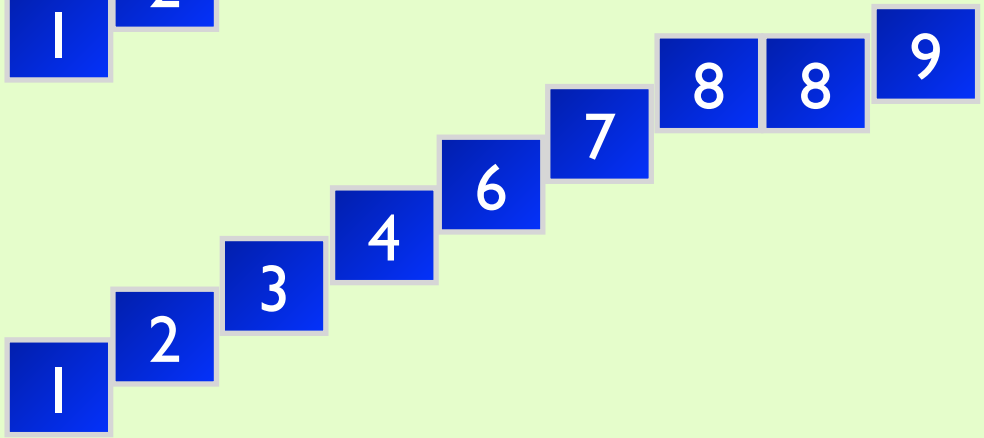
- **Zusammenfassen von wechselnden Folgen**



- **Zusammenfassen von wechselnden Folgen**



- **Zusammenfassen von wechselnden Folgen**



© 2006-2023 Jörg Striegnitz

Bottom-up MergeSort

B Tatsächlicher Ablauf von MergeSort

7 5 3 8 1 4 6 2

- **Umsetzung in Form von Schleifen (keine Rekursion!)**

© 2006-2023 Jörg Striegnitz

Bottom-up `MergeSort`

B Tatsächlicher Ablauf von `MergeSort`

- **Umsetzung in Form von Schleifen (keine Rekursion!)**
- **Ablauf entspricht „Breitendurchlauf“**

© 2006-2023 Jörg Striegnitz

Bottom-up MergeSort

B Tatsächlicher Ablauf von MergeSort

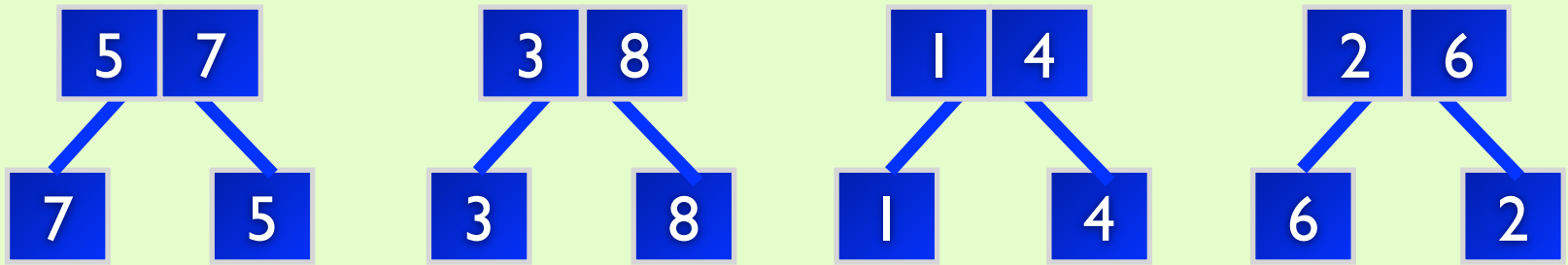
7 5 3 8 1 4 6 2

- Umsetzung in Form von Schleifen (keine Rekursion!)
- Ablauf entspricht „Breitendurchlauf“
- Fazit: schlechtere Speicherlokalität für große (!) Felder

© 2006-2023 Jörg Striegnitz

Bottom-up MergeSort

B Tatsächlicher Ablauf von MergeSort

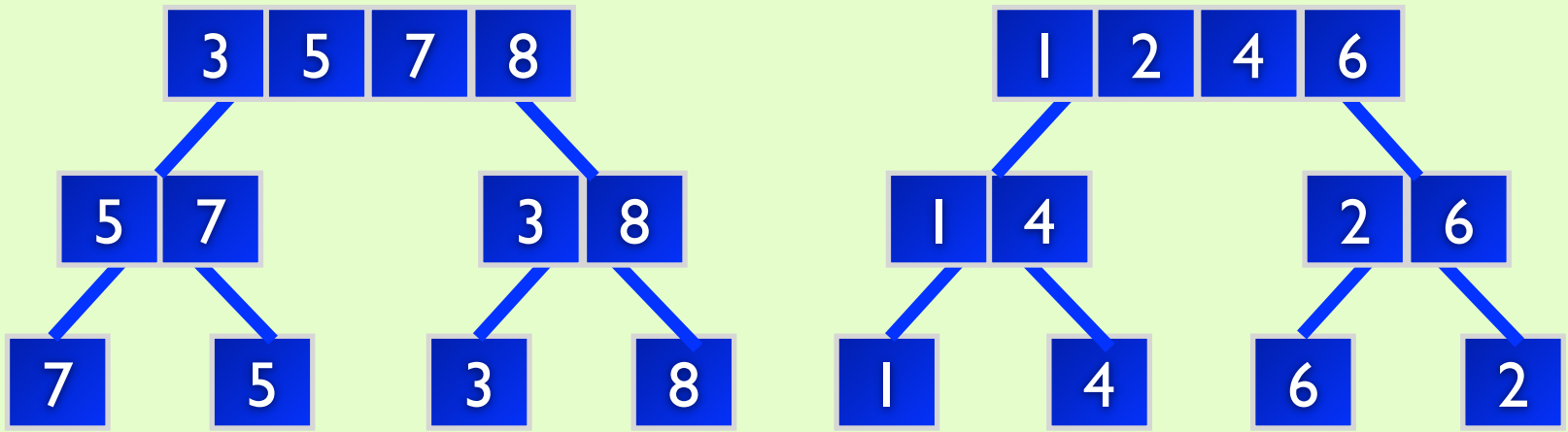


- Umsetzung in Form von Schleifen (keine Rekursion!)
- Ablauf entspricht „Breitendurchlauf“
- Fazit: schlechtere Speicherlokalität für große (!) Felder

© 2006-2023 Jörg Striegnitz

Bottom-up MergeSort

B Tatsächlicher Ablauf von MergeSort

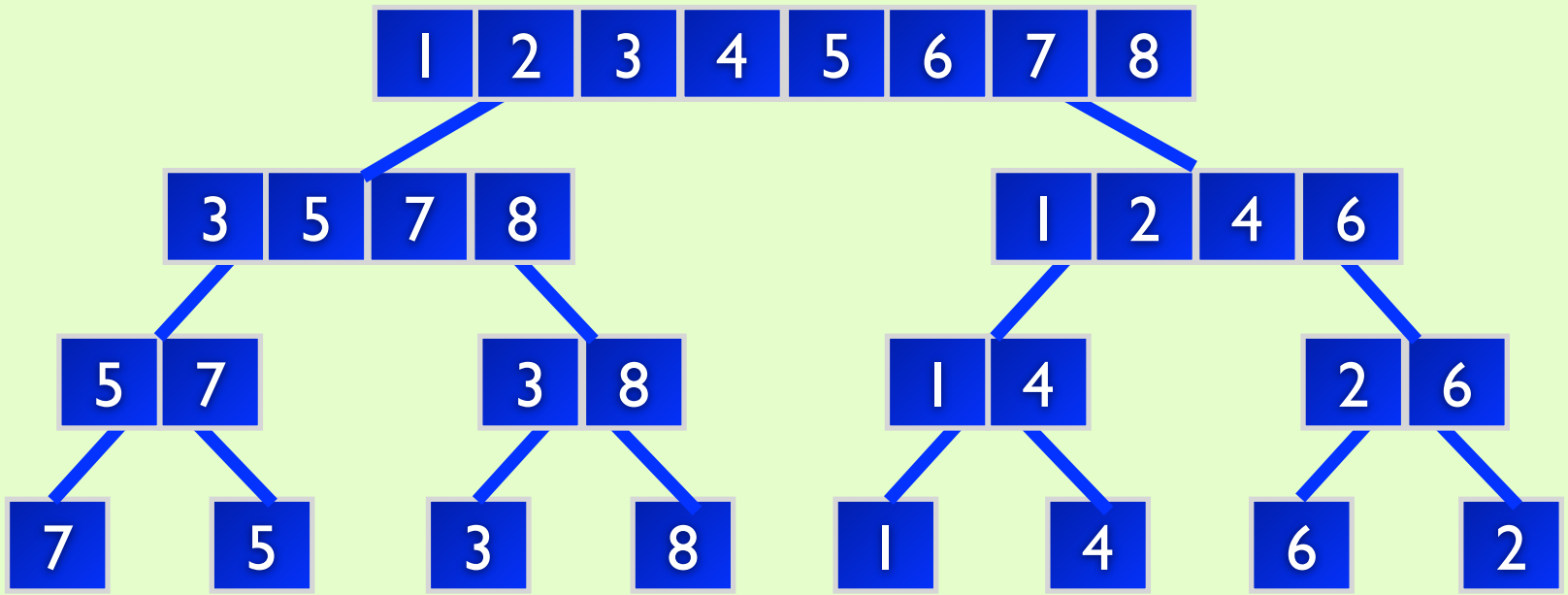


- Umsetzung in Form von Schleifen (keine Rekursion!)
- Ablauf entspricht „Breitendurchlauf“
- Fazit: schlechtere Speicherlokalität für große (!) Felder

© 2006-2023 Jörg Striegnitz

Bottom-up MergeSort

B Tatsächlicher Ablauf von MergeSort



- Umsetzung in Form von Schleifen (keine Rekursion!)
- Ablauf entspricht „Breitendurchlauf“
- Fazit: schlechtere Speicherlokalität für große (!) Felder

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
- 4.1.1. Selection Sort
- 4.1.2. Insertion Sort
- 4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
- 4.2.1. Quicksort
- 4.2.2. Heapsort
- 4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
- 4.3.1. Counting Sort
- 4.3.2. Bucket Sort
- 4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Untergrenze: Zahl der Vergleiche

- **Bisher: Algorithmen basieren auf Schlüsselvergleichen**
  - Wie viele Vergleiche müssen **mindestens** erfolgen?
  - Ist $n \cdot \log n$ das Limit oder geht es besser?

© 2006-2023 Jörg Striegnitz

Untergrenze: Zahl der Vergleiche

- **Bisher: Algorithmen basieren auf Schlüsselvergleichen**
  - Wie viele Vergleiche müssen **mindestens** erfolgen?
  - Ist $n \cdot \log n$ das Limit oder geht es besser?

S Untere Grenze: Vergleichsbasierte Sortierverfahren

Ein auf Schlüsselvergleichen basierender Sortier-Algorithmus benötigt im **schlimmsten Fall** mindestens

$$\Omega\left(n \cdot \mathrm{ld}(n)\right)$$

viele Schlüsselvergleiche.

© 2006-2023 Jörg Striegnitz

Untergrenze: Zahl der Vergleiche

- **Bisher: Algorithmen basieren auf Schlüsselvergleichen**
  - Wie viele Vergleiche müssen **mindestens** erfolgen?
  - Ist $n \cdot \log n$ das Limit oder geht es besser?

S Untere Grenze: Vergleichsbasierte Sortierverfahren

Ein auf Schlüsselvergleichen basierender Sortier-Algorithmus benötigt im **schlimmsten Fall** mindestens

$$\Omega\left(n \cdot \mathrm{ld}(n)\right)$$

viele Schlüsselvergleiche.

- **Algorithmus mit weniger Schlüsselvergleichen nicht möglich!**

- **Beweisidee:**
  - Algorithmus für Feldgröße n als Entscheidungsbaum mit $n!$ Blättern
  - Jeder vergleichsbasierte Algorithmus lässt sich so darstellen

© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 1

- **Wir betrachten exemplarisch *Insertion Sort***
  - *Wir protokollieren mögliche Abläufe in einem **Entscheidungsbaum***
  - *Sortiert wird indirekt!*

A	A0	A1	A2

π	0	1	2

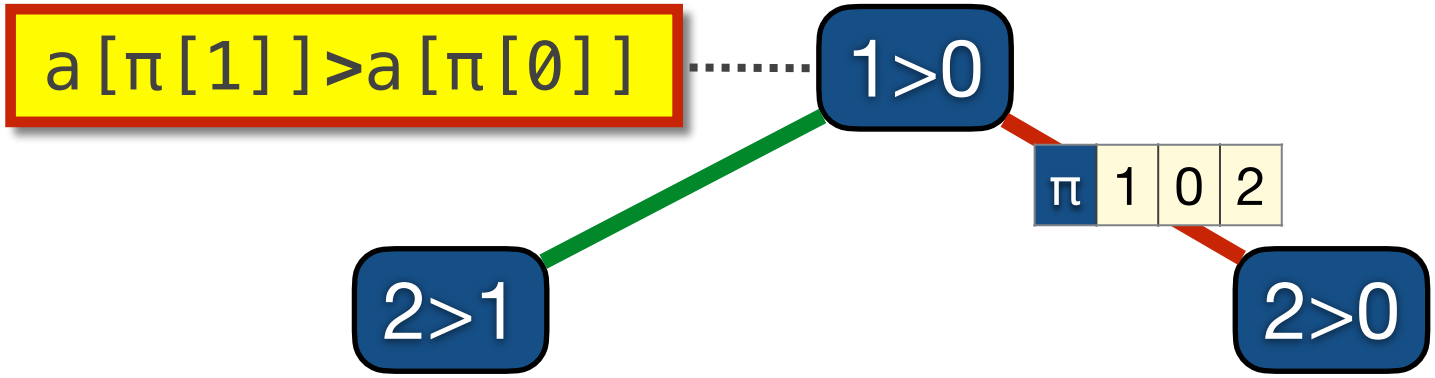
© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 1

- **Wir betrachten exemplarisch *Insertion Sort***
  - *Wir protokollieren mögliche Abläufe in einem **Entscheidungsbaum***
  - *Sortiert wird indirekt!*

A	A0	A1	A2

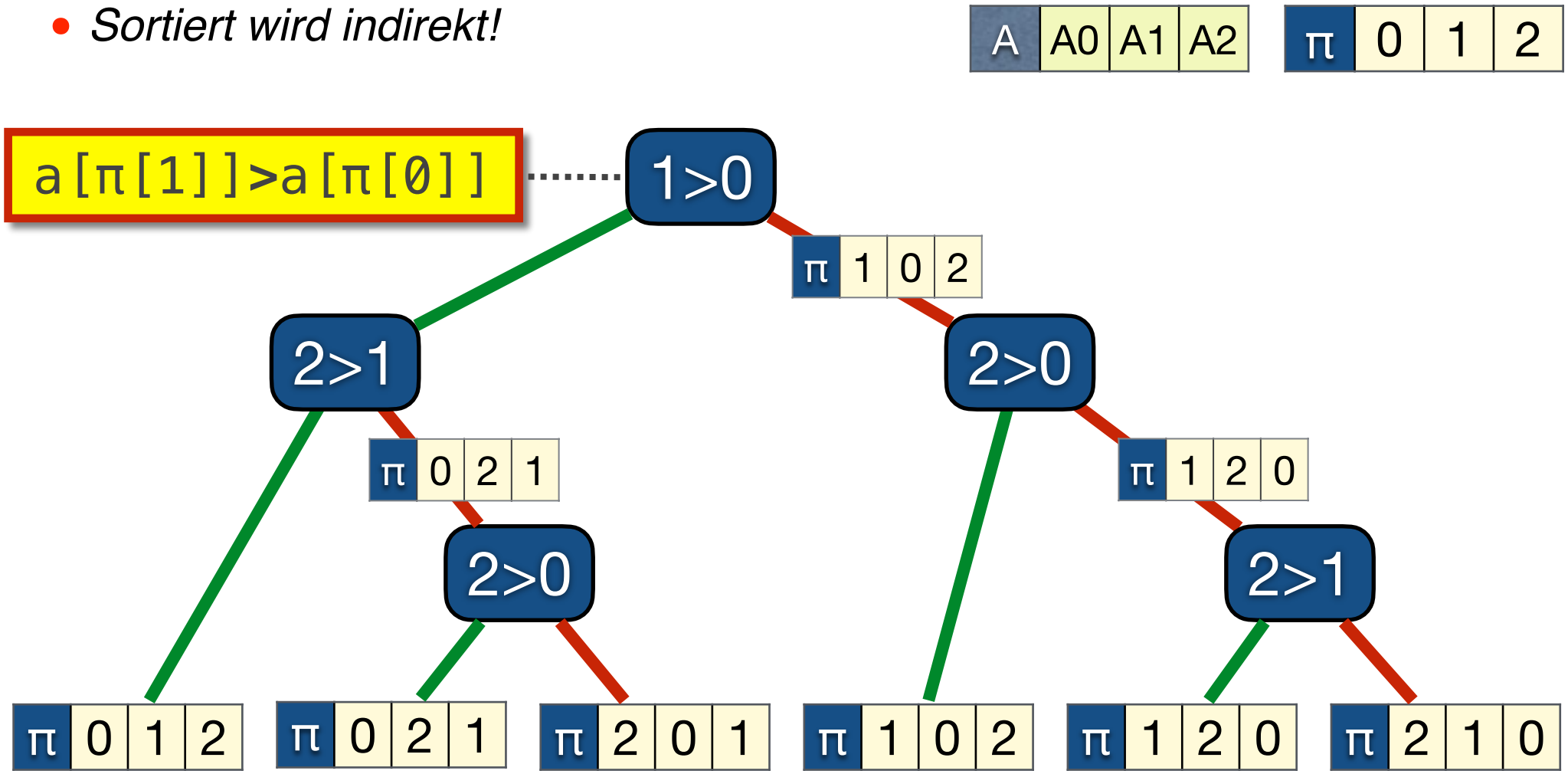
π	0	1	2



© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 1

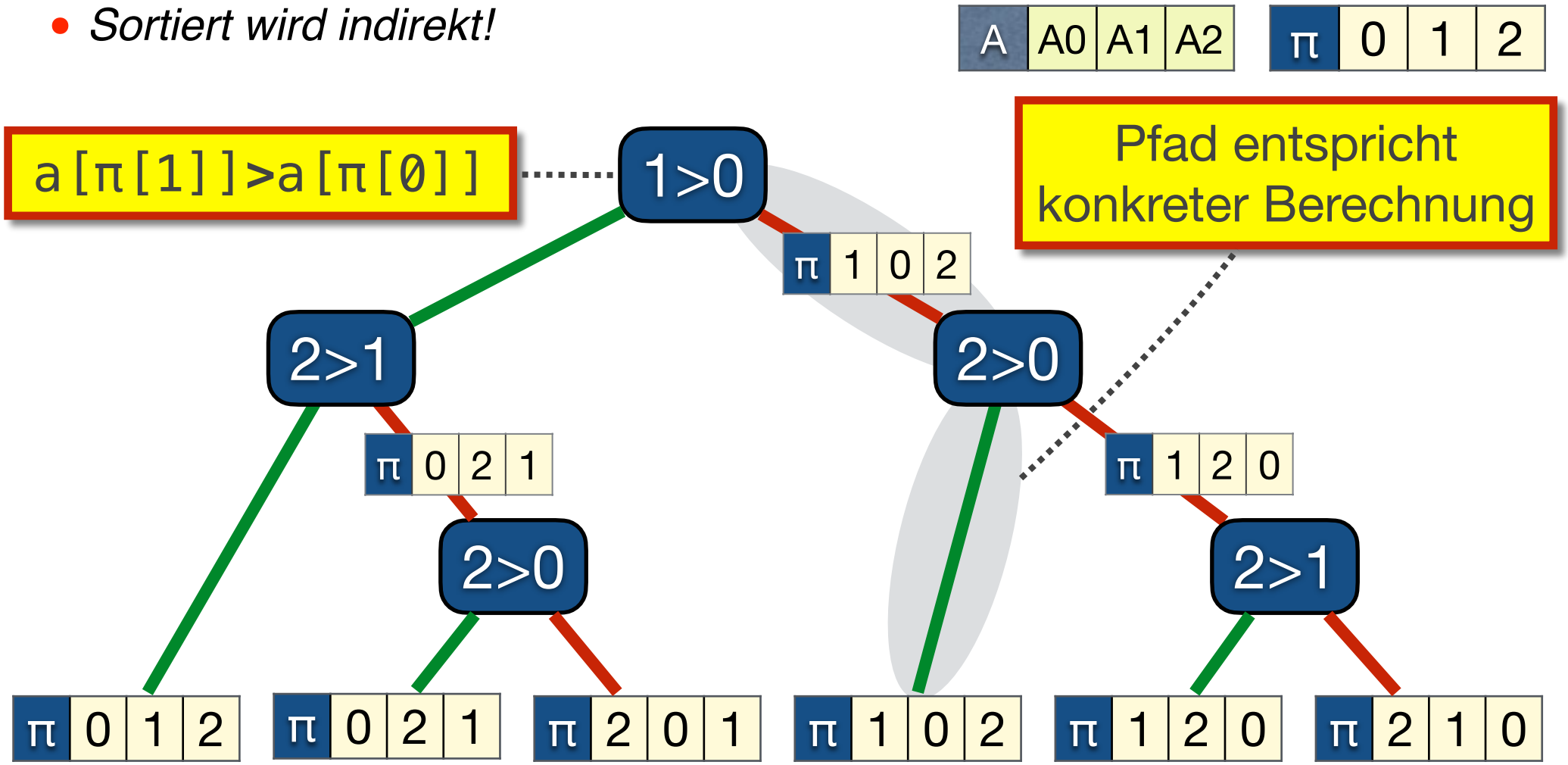
- **Wir betrachten exemplarisch *Insertion Sort***
  - *Wir protokollieren mögliche Abläufe in einem **Entscheidungsbaum***
  - *Sortiert wird indirekt!*



© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 1

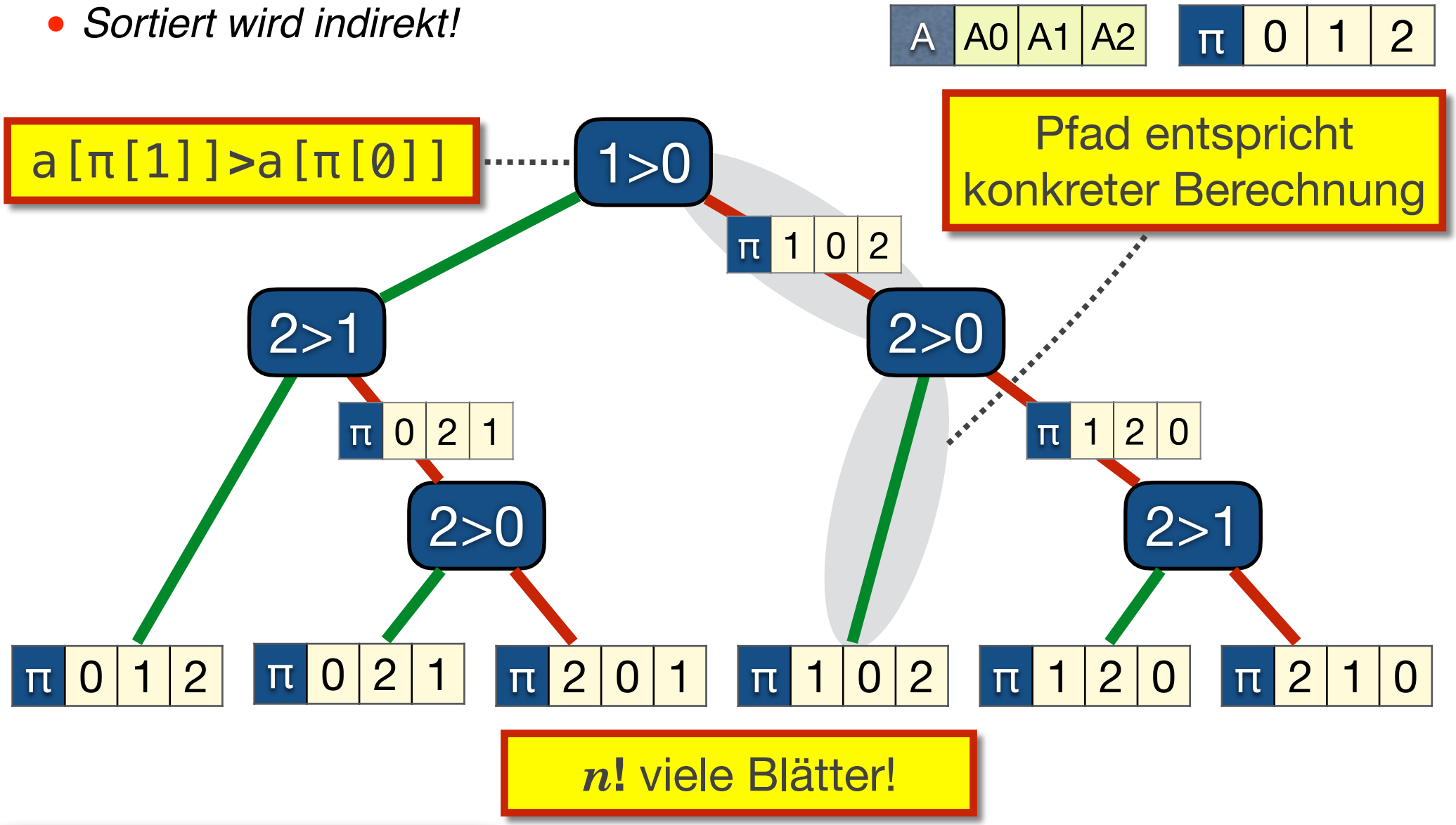
- **Wir betrachten exemplarisch *Insertion Sort***
  - *Wir protokollieren mögliche Abläufe in einem **Entscheidungsbaum***
  - *Sortiert wird indirekt!*



© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 1

- **Wir betrachten exemplarisch *Insertion Sort***
  - *Wir protokollieren mögliche Abläufe in einem **Entscheidungsbaum***
  - *Sortiert wird indirekt!*



© 2006-2023 Jörg Striegnitz

Sortier-Algorithmen als Entscheidungsbäume - 2

- ***Worst Case* Vergleiche = Höhe des Entscheidungsbaums**

- **Zahl der Blätter ist unabhängig vom konkreten Algorithmus**

Ein Entscheidungsbaum für einen vergleichsbasierenden Sortier-Algorithmus, der auf n Elemente angewendet wird, ist ein Binärbaum mit $n!$ Blättern (für jede Permutation ein Blatt).
Die minimale Höhe dieses Baums ist $\mathsf{ld}(n!)$; nun ist einerseits

$$\begin{array}{rrcl} & n! & \leq & n^n \\ \Leftrightarrow & n! & \leq & 2^{\mathsf{ld}(n^n)} \\ \Leftrightarrow & n! & \leq & 2^{n \cdot \mathsf{ld}(n)} \\ \Leftrightarrow & \mathsf{ld}(n!) & \leq & \mathsf{ld}(2^{n \cdot \mathsf{ld}(n)}) \\ \Leftrightarrow & \mathsf{ld}(n!) & \leq & n \cdot \mathsf{ld}(n) \end{array}$$

und andererseits

$$\begin{array}{rrcl} & n! & \geq & \frac{n}{2}^{\frac{n}{2}} \\ \Leftrightarrow & \mathsf{ld}(n!) & \geq & \frac{n}{2} \cdot \mathsf{ld}(\frac{n}{2}) \end{array}$$

und somit ist die minimale Höhe des Baumes in $\Theta(n \cdot \mathsf{ld}(n))$!

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
4.1.1. Selection Sort
4.1.2. Insertion Sort
4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
4.2.1. Quicksort
4.2.2. Heapsort
4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
4.3.1. Counting Sort
4.3.2. Bucket Sort
4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Counting Sort

- **Auch bekannt als:** *Bin Sorting*, *Distribution Counting*, Sortieren mittels Histogramm, *KSort*

© 2006-2023 Jörg Striegnitz

Counting Sort

- **Auch bekannt als:** *Bin Sorting*, *Distribution Counting*, Sortieren mittels Histogramm, *KSort*

- **Voraussetzung:** Schlüssel können als ganzzahlige Werte im Bereich $0,\ldots,K\text{-}1$ dargestellt werden, so dass sie als **Feld-Index** verwendet werden können - **K möglichst klein!**

© 2006-2023 Jörg Striegnitz

Counting Sort

- **Auch bekannt als:** *Bin Sorting*, *Distribution Counting*, Sortieren mittels Histogramm, *KSort*

- **Voraussetzung:** Schlüssel können als ganzzahlige Werte im Bereich $0,\ldots,K\text{-}1$ dargestellt werden, so dass sie als **Feld-Index** verwendet werden können - **K möglichst klein!**

- **Prinzip:**

  1. Erstelle ein **Histogramm**; d.h. zähle für jeden Schlüsselwert wie häufig er vorkommt
  2. Berechne aus dem Histogramm die Position für jeden Record
     Die Anzahl der Elemente kleiner oder gleich einem Schlüssel
  3. Bewege die Records (mit rückläufigem Index) an ihre errechnete Position

© 2006-2023 Jörg Striegnitz

B Counting Sort

© 2006-2023 Jörg Striegnitz

B Counting Sort

- Angenommen, wir wollten ein Feld, in dem nur Großbuchstaben gespeichert sind, sortieren.

A	A	B	A	C	A	D	A	B	B	A	D	D	A

© 2006-2023 Jörg Striegnitz

B Counting Sort

- Angenommen, wir wollten ein Feld, in dem nur Großbuchstaben gespeichert sind, sortieren.

A	A	B	A	C	A	D	A	B	B	A	D	D	A

- Die Voraussetzung zur Anwendung von *Counting Sort* lässt sich leicht erfüllen, indem wir Großbuchstaben wie folgt auf Indizes abbilden:

$$
\begin{aligned}
A &\rightarrow 0 \\
B &\rightarrow 1 \\
C &\rightarrow 2 \\
&\vdots \\
Z &\rightarrow 25
\end{aligned}
$$

© 2006-2023 Jörg Striegnitz

Counting Sort: Histogramm 1

B Counting Sort

- Jetzt können wir leicht ein **Histogramm** erstellen:

A	B	B	A	C	A	D	A	B	B	A	D	D	A

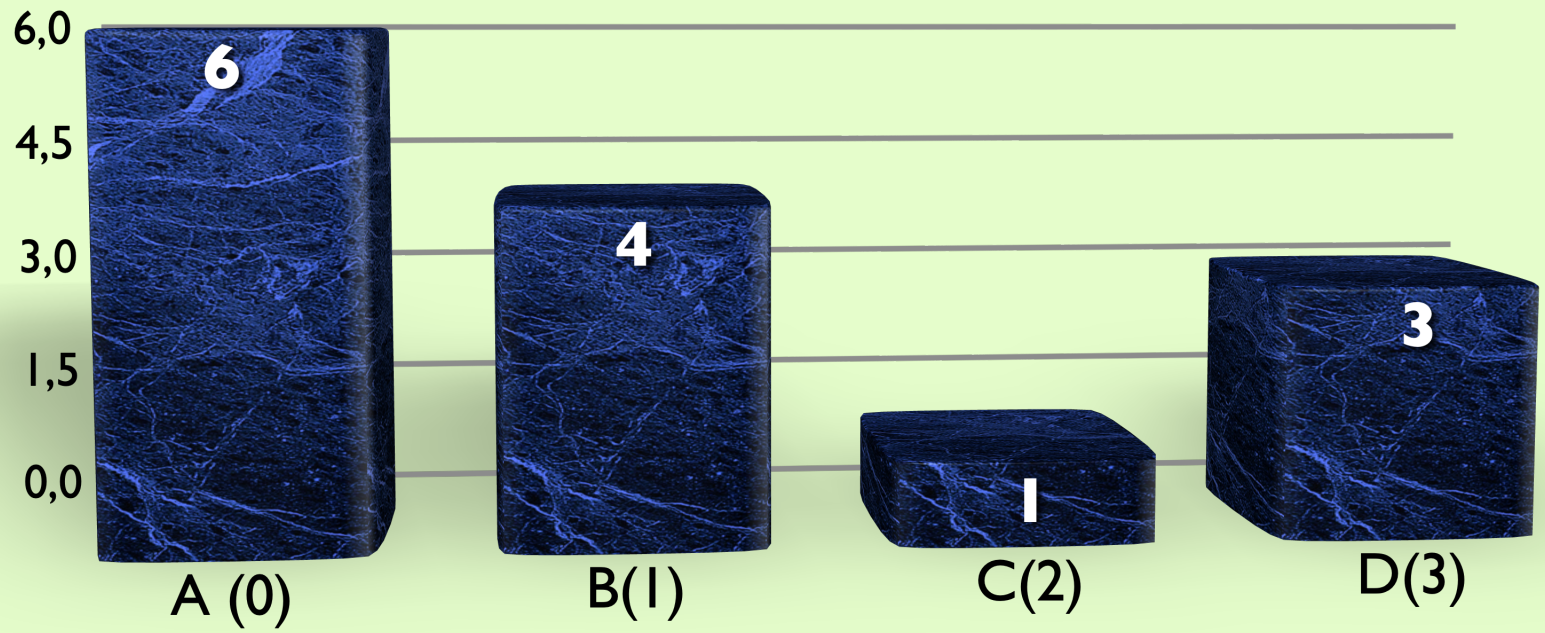
© 2006-2023 Jörg Striegnitz

Counting Sort: Histogramm 1

B Counting Sort

- Jetzt können wir leicht ein **Histogramm** erstellen:

A	B	B	A	C	A	D	A	B	B	A	D	D	A



© 2006-2023 Jörg Striegnitz

Counting Sort: Histogram 2

B Counting Sort

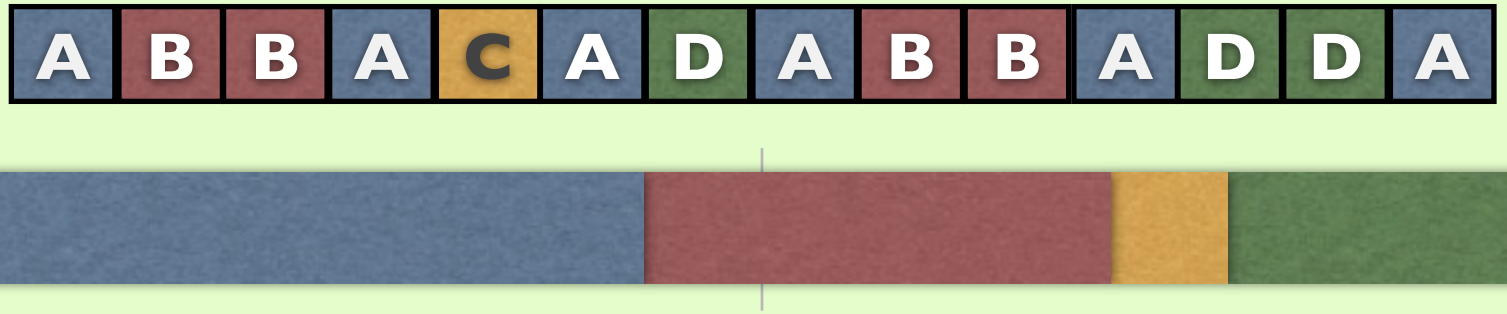
- Am **Histogramm** kann man „ablesen“, wie das sortierte Feld aussieht:

© 2006-2023 Jörg Striegnitz

Counting Sort: Histogram 2

B Counting Sort

- Am **Histogramm** kann man „ablesen“, wie das sortierte Feld aussieht:

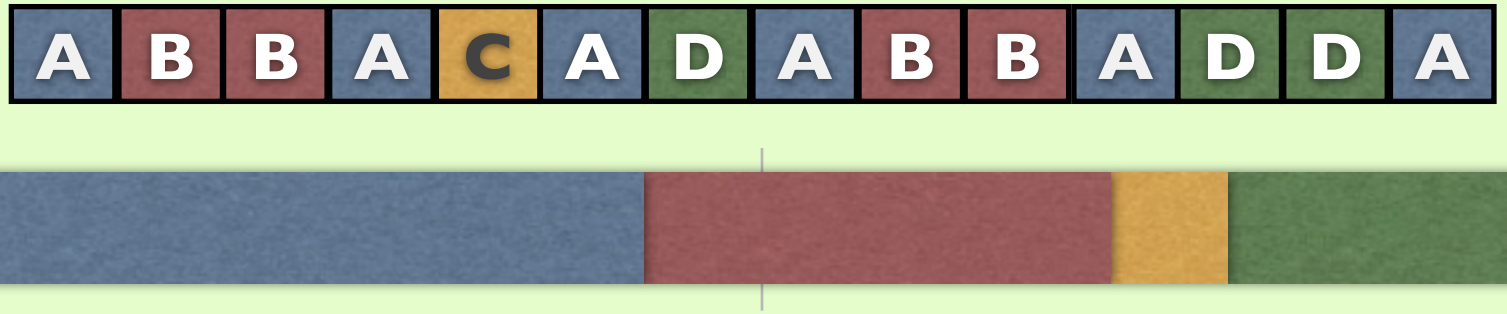


© 2006-2023 Jörg Striegnitz

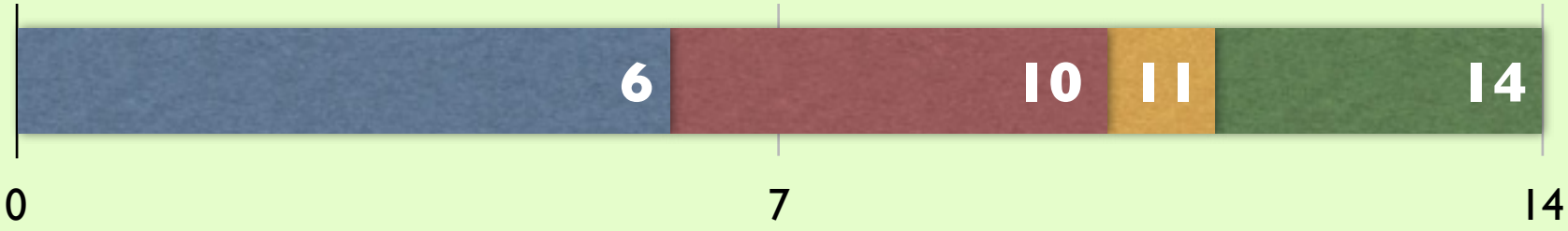
Counting Sort: Histogram 2

B Counting Sort

- Am **Histogramm** kann man „ablesen“, wie das sortierte Feld aussieht:



- Berechne aus Histogramm die größten **Offsets**

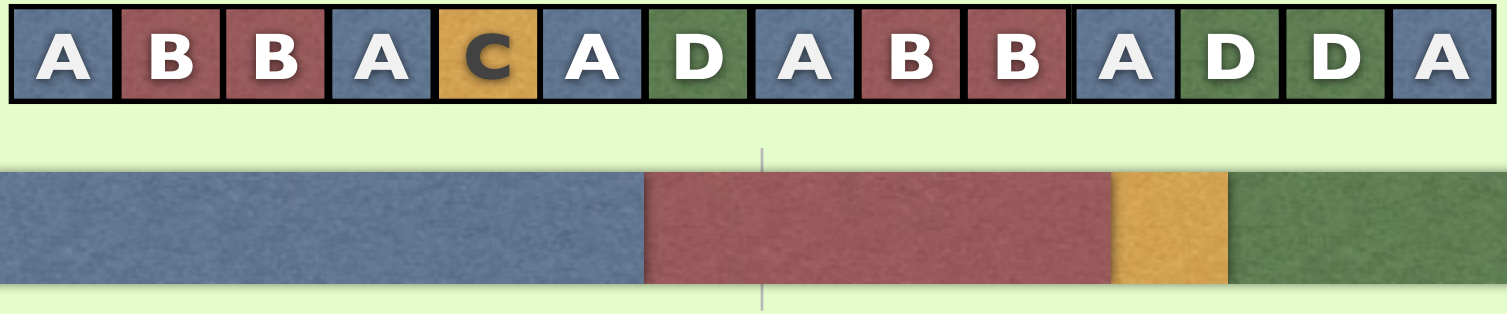


© 2006-2023 Jörg Striegnitz

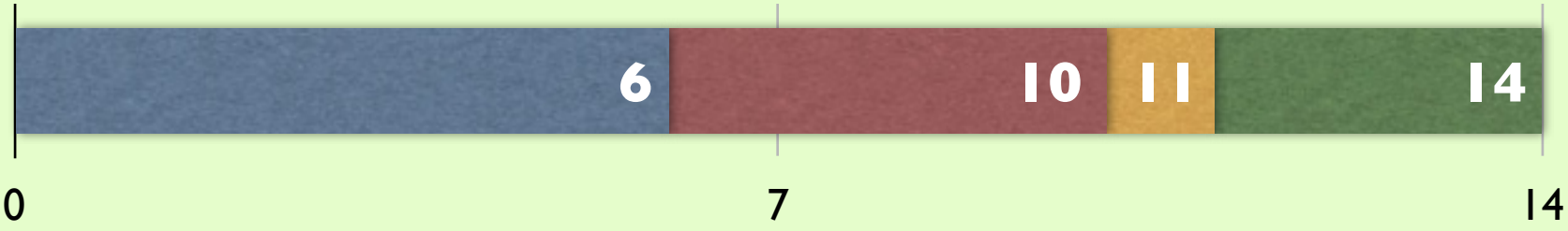
Counting Sort: Histogram 2

B Counting Sort

- Am **Histogramm** kann man „ablesen“, wie das sortierte Feld aussieht:



- Berechne aus Histogramm die größten **Offsets**



- Durchlaufe Feld **von hinten nach vorne**, befülle neues Feld mithilfe der Offsets; passe Offsets entsprechend an (verringern).

© 2006-2023 Jörg Striegnitz

Implementierung (JAVA)

```
public static void CountingSort(int feld[]) {
    // Vereinfachende Annahme: Wertebereich = Indexmenge
    int histogram[] = new int[feld.length];
    int hilfsFeld[] = new int[feld.length];
    // Histogramm berechnen:
    for (int i=0 ; i<feld.length ; ++i)
      histogram[feld[i]]++;
    // Zielpositionen berechnen ...
    for (int i=1 ; i<histogram.length ; ++i)
      histogram[i]=histogram[i-1] + histogram[i];
    // Das sortierte Feld in Hilfsfeld erzeugen ...
    for (int i=feld.length-1 ; i>=0 ; i--) {
      hilfsFeld[histogram[feld[i]]-1]=feld[i];
      histogram[feld[i]]--;
    }
    // ... und in Ausgangsfeld kopieren
    for (int i=0 ; i<feld.length ; ++i)
      feld[i]=hilfsFeld[i];
}
```


© 2006-2023 Jörg Striegnitz

Implementierung (JAVA)

```
public static void CountingSort(int feld[]) {
    // Vereinfachende Annahme: Wertebereich = Indexmenge
    int histogram[] = new int[feld.length];
    int hilfsFeld[] = new int[feld.length];
    // Histogramm berechnen:
    for (int i=0 ; i<feld.length ; ++i)
      histogram[feld[i]]++;
    // Zielpositionen berechnen ...
    for (int i=1 ; i<histogram.length ; ++i)
      histogram[i]=histogram[i-1] + histogram[i];
    // Das sortierte Feld in Hilfsfeld erzeugen ...
    for (int i=feld.length-1 ; i>=0 ; i--) {
      hilfsFeld[histogram[feld[i]]-1]=feld[i];
      histogram[feld[i]]--;
    }
    // ... und in Ausgangsfeld kopieren
    for (int i=0 ; i<feld.length ; ++i)
      feld[i]=hilfsFeld[i];
}
```

```
histogram[0]=#A
histogram[1]=#A+#B
histogram[2]=#A+#B+#C
...
```


© 2006-2023 Jörg Striegnitz

Eigenschaften von Counting Sort

- **Wegen des rückläufigen Index ist *Counting Sort* stabil**

© 2006-2023 Jörg Striegnitz

Eigenschaften von Counting Sort

- **Wegen des rückläufigen Index ist *Counting Sort* stabil**
- **Die hier vorgestellte Variante arbeitet nicht in situ**
  Eine *in situ*-Variante ist möglich, allerdings verliert man die Stabilität

© 2006-2023 Jörg Striegnitz

Eigenschaften von Counting Sort

- **Wegen des rückläufigen Index ist *Counting Sort* stabil**
- **Die hier vorgestellte Variante arbeitet nicht in situ**
  Eine *in situ*-Variante ist möglich, allerdings verliert man die Stabilität
- **Für die Zeit- und Platzkomplexität gilt**

$$\begin{aligned} T(n) &= O(N+K) \\ &= O(max\{N,K\}) \end{aligned}$$

© 2006-2023 Jörg Striegnitz

Eigenschaften von Counting Sort

- **Wegen des rückläufigen Index ist *Counting Sort* stabil**

- **Die hier vorgestellte Variante arbeitet nicht in situ**
  Eine *in situ*-Variante ist möglich, allerdings verliert man die Stabilität

- **Für die Zeit- und Platzkomplexität gilt**

$$\begin{aligned} T(n) &= O(N+K) \\ &= O(max\{N,K\}) \end{aligned}$$

- **Beachte**: Speicherbedarf Histogramm anhängig von K !!!
  Folge: ist das größte Element z.B. in $O(n^3)$, ist die Laufzeitkomplexität schlechter als bei den einfachen Verfahren!

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
4.1.1. Selection Sort
4.1.2. Insertion Sort
4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
4.2.1. Quicksort
4.2.2. Heapsort
4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
4.3.1. Counting Sort
4.3.2. Bucket Sort
4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Bucket Sort

- **Auch bekannt als**: Sortieren durch Fachverteilen

© 2006-2023 Jörg Striegnitz

Bucket Sort

- **Auch bekannt als**: Sortieren durch Fachverteilen

- **Voraussetzungen:**
  - Schlüssel als ganze Zahlen aus dem Bereich $0,\ldots,K\text{-}1$
    wie *Counting Sort*, aber K **nicht zwingend klein**
  - Alle Werte sind für zu sortierende Eingabe gleich wahrscheinlich
    diskrete Gleichverteilung der Werte

© 2006-2023 Jörg Striegnitz

Bucket Sort

- **Auch bekannt als**: Sortieren durch Fachverteilen

- **Voraussetzungen:**
  - Schlüssel als ganze Zahlen aus dem Bereich $0,\ldots,K\text{-}1$
    wie *Counting Sort*, aber K **nicht zwingend klein**
  - Alle Werte sind für zu sortierende Eingabe gleich wahrscheinlich
    diskrete Gleichverteilung der Werte

- **Idee**:
  - Sortiere die Schlüssel in B Fächer (*Buckets*)
    Schlüssel X kommt in Fach $\lfloor X/(K/B) \rfloor$; Fächer als verkettete Liste
  - Sortiere Schlüssel eines Faches mit *Insertion Sort*
    Insertion Sort lässt sich einfach auf verkettete Listen übertragen -> sortiert wird oft bereits beim Einfügen!
  - Konkateniere die Listen der einzelnen Fächer zum Ergebnis

© 2006-2023 Jörg Striegnitz

Bucket Sort (Beispiel)

B Bucket Sort

- Seien K=100 und B=10

A	99	55	42	33	23	21	9	13	87	32	56	40	74	62	57	12	1	91	47	7

- **Phase 1**: **Verteilen der Schlüssel auf die Fächer**

0	●
1	●
2	●
3	●
4	●
5	●
6	●
7	●
8	●
9	●

© 2006-2023 Jörg Striegnitz

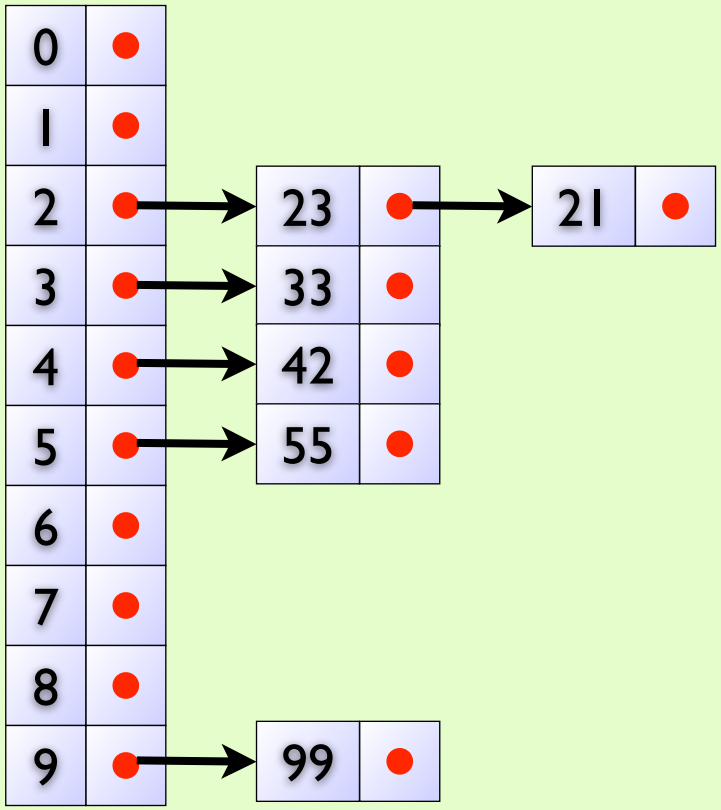
Bucket Sort (Beispiel)

B Bucket Sort

- Seien K=100 und B=10

A	99	55	42	33	23	21	9	13	87	32	56	40	74	62	57	12	1	91	47	7

- **Phase 1**: **Verteilen der Schlüssel auf die Fächer**



© 2006-2023 Jörg Striegnitz

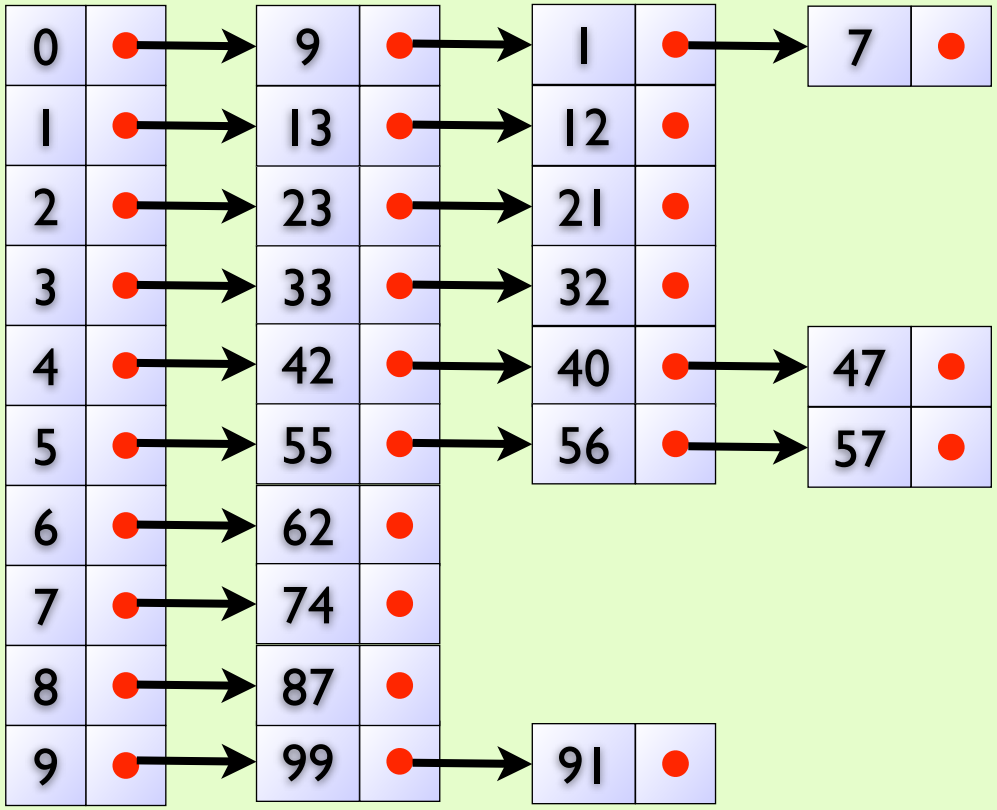
Bucket Sort (Beispiel)

B Bucket Sort

- Seien K=100 und B=10

A	99	55	42	33	23	21	9	13	87	32	56	40	74	62	57	12	1	91	47	7

- **Phase 1**: **Verteilen der Schlüssel auf die Fächer**



© 2006-2023 Jörg Striegnitz

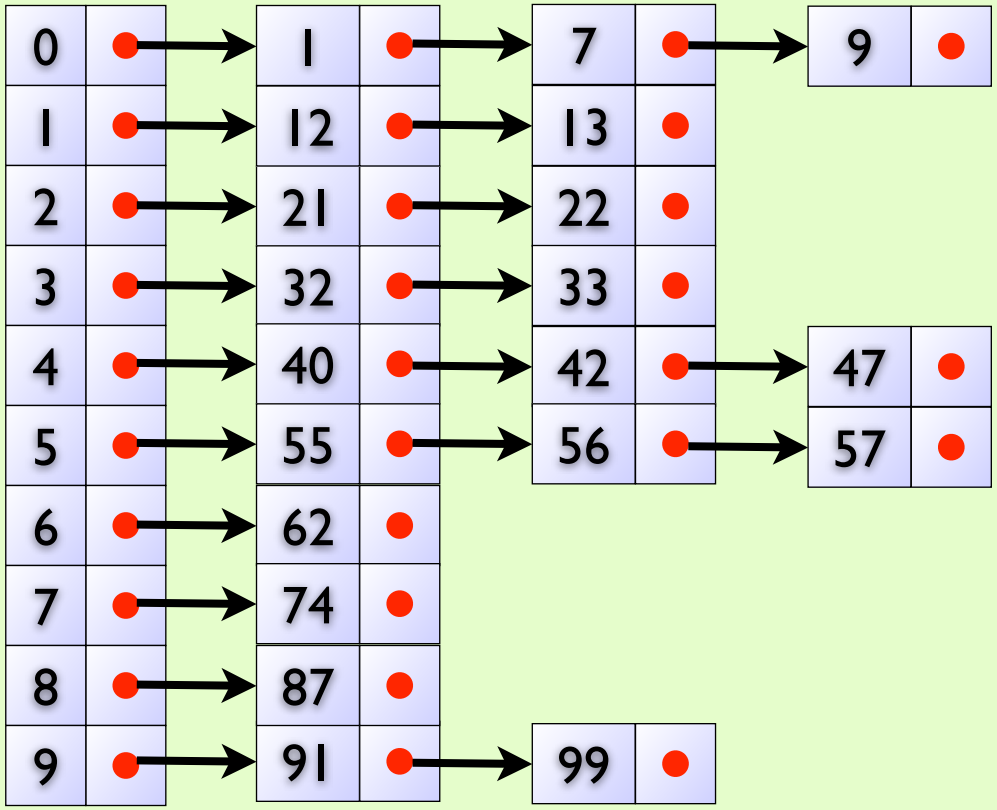
Bucket Sort (Beispiel)

B Bucket Sort

- Seien K=100 und B=10

A	99	55	42	33	23	21	9	13	87	32	56	40	74	62	57	12	1	91	47	7

- **Phase 2**: **Sortieren der Fächer mit *Insertion Sort***

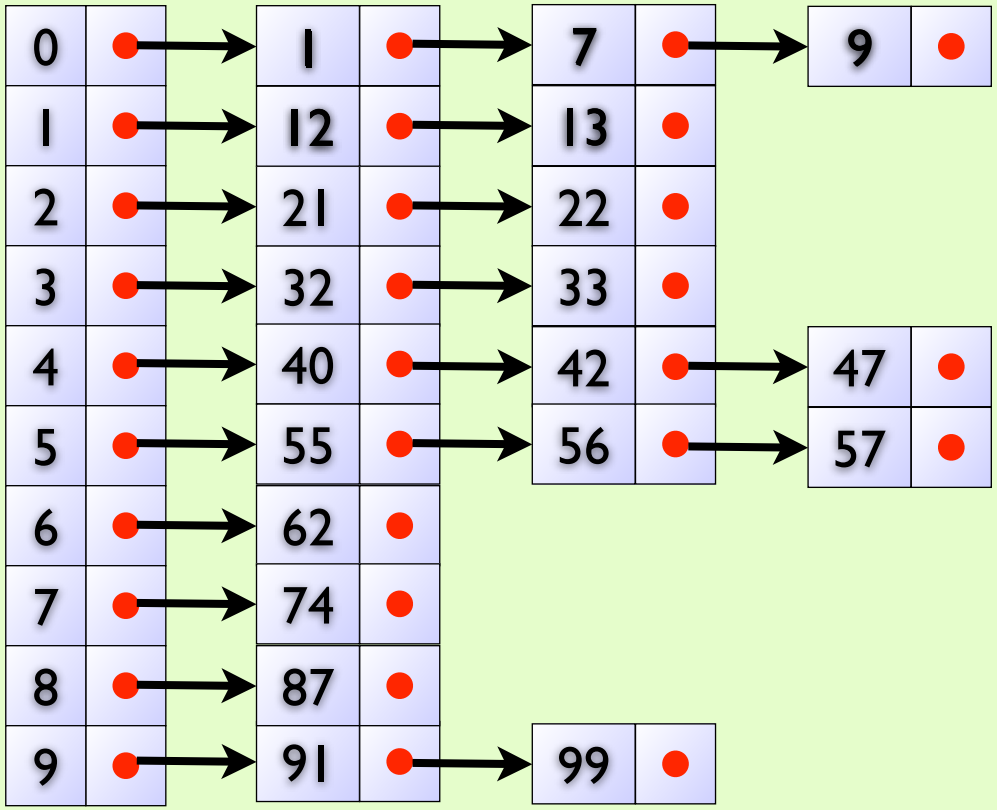


© 2006-2023 Jörg Striegnitz

Bucket Sort (Beispiel)

B Bucket Sort

- **Phase 3**: Konkatenation der Teilfolgen



Phasen 2 und 3 laufen typischerweise verzahnt!

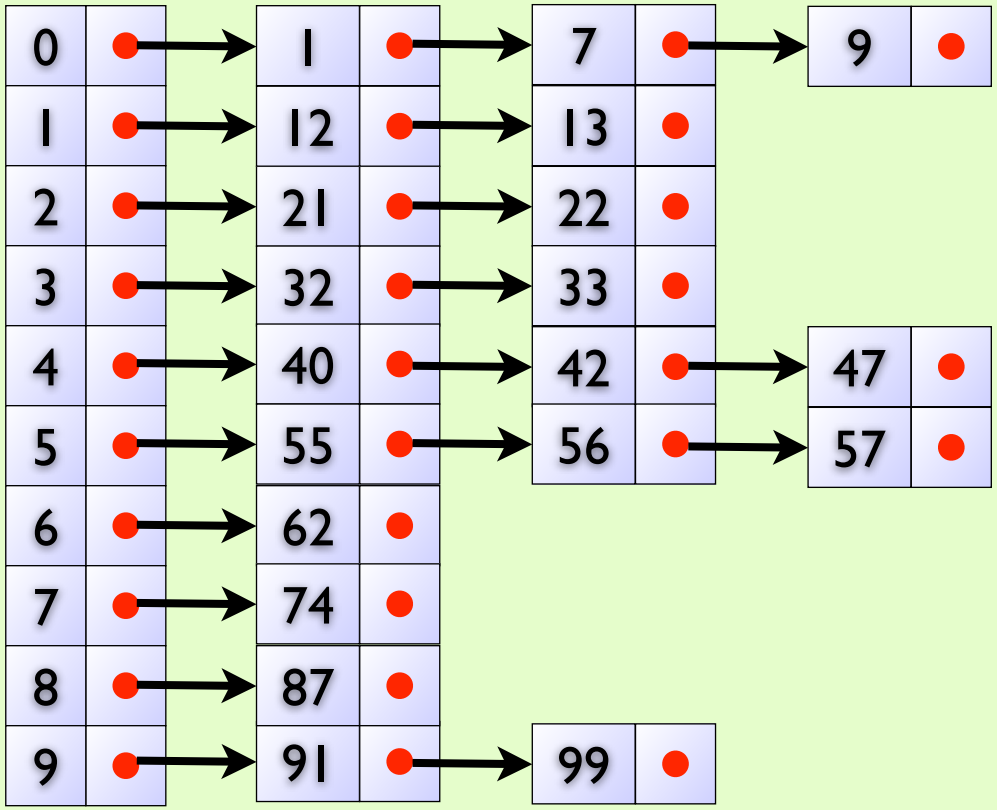
R																				

© 2006-2023 Jörg Striegnitz

Bucket Sort (Beispiel)

B Bucket Sort

- **Phase 3**: Konkatenation der Teilfolgen



Phasen 2 und 3 laufen typischerweise verzahnt!

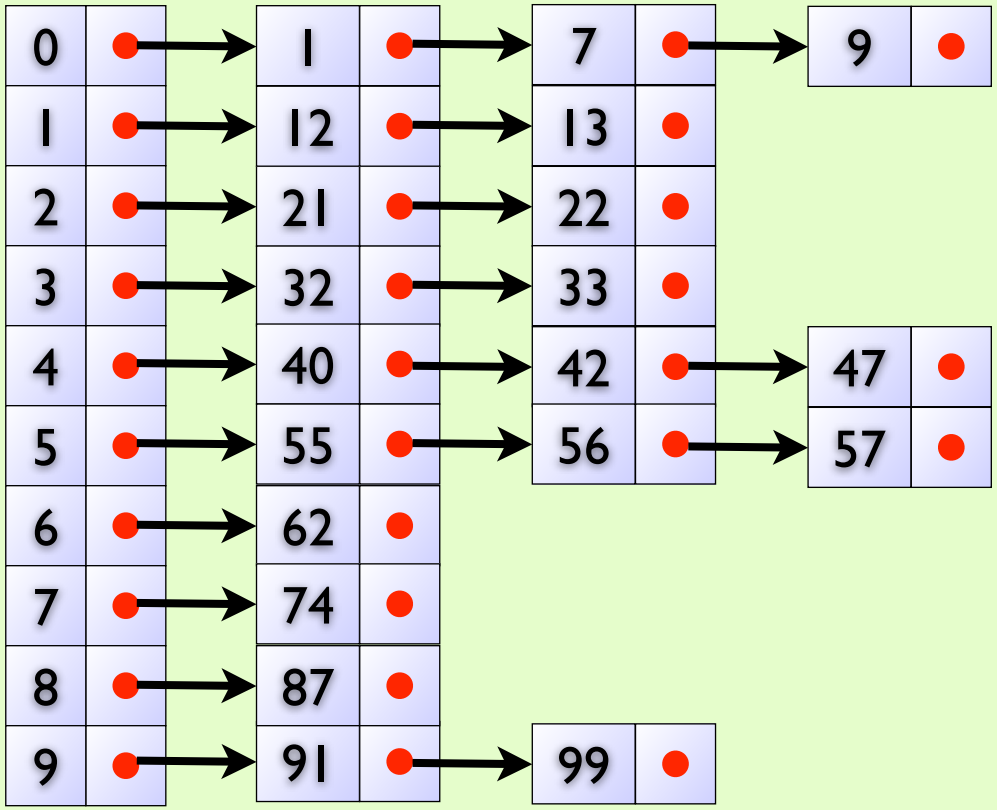
R	1	7	9	12	13															

© 2006-2023 Jörg Striegnitz

Bucket Sort (Beispiel)

B Bucket Sort

- **Phase 3**: Konkatenation der Teilfolgen



Phasen 2 und 3 laufen typischerweise verzahnt!

R	1	7	9	12	13	21	22	32	33	40	42	47	55	56	57	61	74	87	91	99

© 2006-2023 Jörg Striegnitz

Bucket Sort (JAVA - SortedList)

```
public class SortedList {
 public class Node {
    public int data;
    public Node next;
    public Node(int data) { this.data = data; this.next = null; }
  }
  public Node head = null;

  // Wie das "Aufnehmen einer Karte" bei Insertion Sort
  void insert(int key) {
    Node it=head,prev=null;
    prev = null;
    while (it != null && it.data <= key) {
      prev = it;
      it   = it.next;
    }
    Node node = new Node(key);
    if (prev == null) {
      node.next = head;
      head = node;
    } else {
      node.next = prev.next;
      prev.next = node;
    }
  }
}
```


© 2006-2023 Jörg Striegnitz

Bucket Sort (JAVA)

```java
public class BucketSort {
  public ArrayList<SortedList> buckets;
  public static int B = 10;
  public static int K = 1000;

  BucketSort() {
    buckets = new ArrayList<SortedList>();
    for (int i = 0; i < B; i++) {
      buckets.add(new SortedList());
    }
  }

  void sort(int[] A) {
    for (int i = 0; i < A.length; i++) {
      buckets.get(A[i] / (K/B)).insert(A[i]);
    }
    int k = 0;
    for (int i = 0; i < B; i++) {
      SortedList.Node it = buckets.get(i).head;
      while (it != null) {
        A[k++] = it.data;
        it = it.next;
      }
    }
  }
}
```


© 2006-2023 Jörg Striegnitz

Eigenschaften von Bucket Sort

- ***Bucket Sort*** **ist** ***stabil*** **aber kein** ***in situ-Verfahren***

© 2006-2023 Jörg Striegnitz

Eigenschaften von Bucket Sort

- ***Bucket Sort* ist *stabil* aber kein *in situ-Verfahren***

- **Platzbedarf geringer als bei *Counting Sort*!**
  B *Buckets* mit je N/B-Elementen (bei angenommener Verteilung)

© 2006-2023 Jörg Striegnitz

Eigenschaften von Bucket Sort

- ***Bucket Sort*** **ist** ***stabil*** **aber kein** ***in situ-Verfahren***

- **Platzbedarf geringer als bei** ***Counting Sort*****!**
  B *Buckets* mit je N/B-Elementen (bei angenommener Verteilung)

- **Zeit-Komplexität wird durch** ***Insertion Sort*** **dominiert**

$$\begin{aligned} T(n) &= O\left(B \cdot (N/B)^2\right) \\ &= O\left((N/B)^2\right) \end{aligned}$$

© 2006-2023 Jörg Striegnitz

Eigenschaften von Bucket Sort

- ***Bucket Sort* ist *stabil* aber kein *in situ-Verfahren***

- **Platzbedarf geringer als bei *Counting Sort*!**
  B *Buckets* mit je N/B-Elementen (bei angenommener Verteilung)

- **Zeit-Komplexität wird durch *Insertion Sort* dominiert**

$$\begin{aligned} T(n) &= O\left(B \cdot (N/B)^2\right) \\ &= O\left((N/B)^2\right) \end{aligned}$$

- **Beachte**: In der Praxis sind die *Buckets* **selten gleichmäßig besetzt**
  Im schlimmsten Fall landen alle Schlüssel in einem Bucket - die Laufzeit liegt dann in $O(n^2)$.

© 2006-2023 Jörg Striegnitz

IV. Sortieren

4. Sortieren

4.1. Einfache Sortierverfahren
4.1.1. Selection Sort
4.1.2. Insertion Sort
4.1.3. Bubble Sort

4.2. Höhere Sortierverfahren
4.2.1. Quicksort
4.2.2. Heapsort
4.2.3. Merge Sort

4.3. Lineare Sortierverfahren
4.3.1. Counting Sort
4.3.2. Bucket Sort
4.3.3. Radix Sort

© 2006-2023 Jörg Striegnitz

Radix Sort

- **Radix:** Basis (D) in einem Stellenwertsystem
  - Binär-, Dezimal- und Hexadezimal am gebräuchlichsten

© 2006-2023 Jörg Striegnitz

Radix Sort

- **Radix:** Basis (D) in einem Stellenwertsystem
  - Binär-, Dezimal- und Hexadezimal am gebräuchlichsten

- **Voraussetzungen:**
  - Schlüssel als Zahlen **zur Basis D** mit fester Länge
    ggf. führende Nullen ergänzen

© 2006-2023 Jörg Striegnitz

Radix Sort

- **Radix:** Basis (D) in einem Stellenwertsystem
  - Binär-, Dezimal- und Hexadezimal am gebräuchlichsten

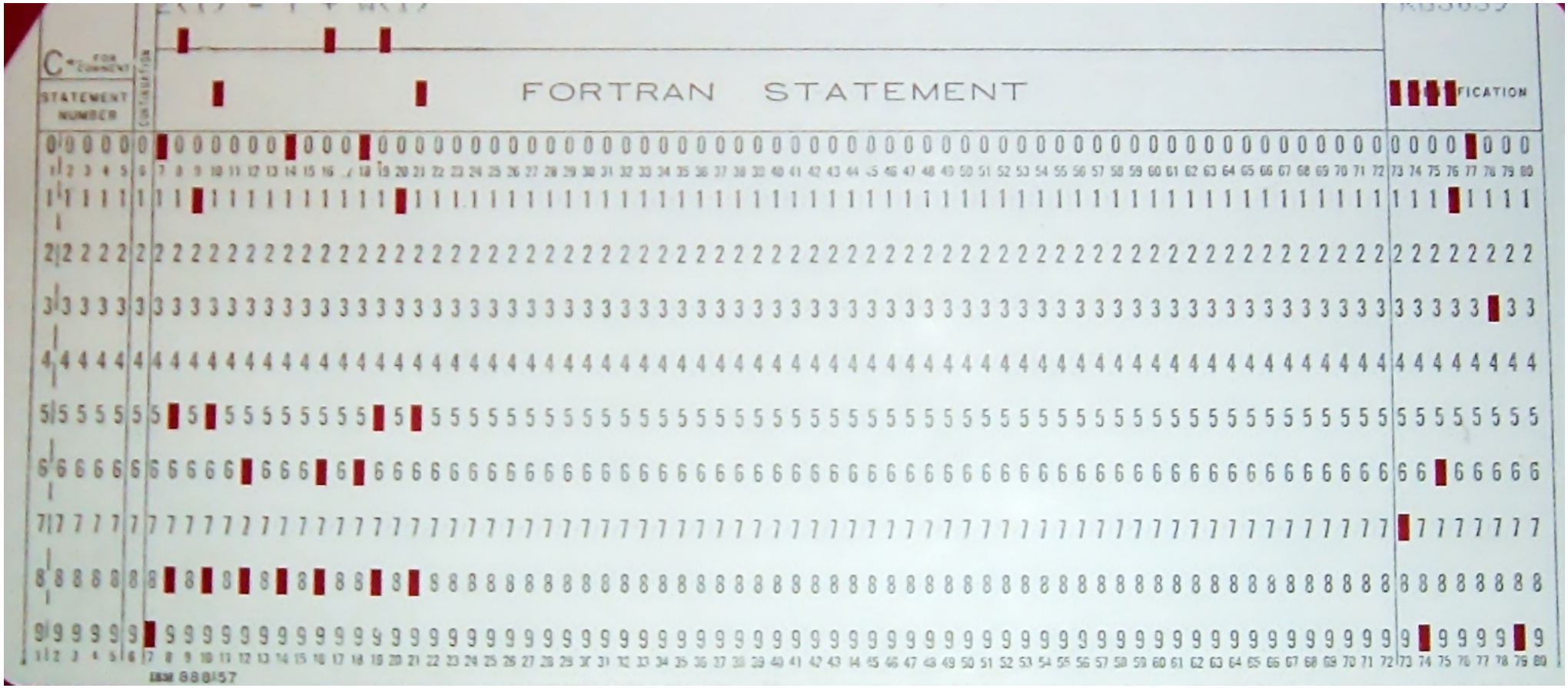
- **Voraussetzungen:**
  - Schlüssel als Zahlen **zur Basis D** mit fester Länge
    ggf. führende Nullen ergänzen

- **Idee**:
  - Sortiere mit ***Counting Sort*** nach der i-ten Ziffern
  - Arbeite von niedrigstwertigen Ziffer zu höchstwertigen Ziffer
    Sortiervorgang ist mehrstufig!
  - Ähnlich dem Sortieren einer Tabelle nach mehreren Spalten

© 2006-2023 Jörg Striegnitz

Historisch

- **Radix-Sort wurde bereits zum Sortieren von Lochkarten verwendet**
  Hierzu gab es spezielle Sortiermaschinen …



© 2006-2023 Jörg Striegnitz

Beispiel: Radix Sort

B Radix Sort mit D=16

0	3	A
F	C	D
1	3	3
C	0	9
3	2	F
A	F	1
9	8	8
3	4	7
D	D	2
7	4	C
E	5	1
C	C	E
E	D	D

© 2006-2023 Jörg Striegnitz

Beispiel: Radix Sort

B Radix Sort mit D=16

0	3	A
F	C	D
1	3	3
C	0	9
3	2	F
A	F	1
9	8	8
3	4	7
D	D	2
7	4	C
E	5	1
C	C	E
E	D	D

A	F	1
E	5	1
D	D	2
1	3	3
3	4	7
9	8	8
C	0	9
0	3	A
7	4	C
F	C	D
E	D	D
C	C	E
3	2	F

© 2006-2023 Jörg Striegnitz

Beispiel: Radix Sort

B Radix Sort mit D=16

0	3	A
F	C	D
1	3	3
C	0	9
3	2	F
A	F	1
9	8	8
3	4	7
D	D	2
7	4	C
E	5	1
C	C	E
E	D	D

A	F	1
E	5	1
D	D	2
1	3	3
3	4	7
9	8	8
C	0	9
0	3	A
7	4	C
F	C	D
E	D	D
C	C	E
3	2	F

C	0	9
3	2	F
1	3	3
0	3	A
3	4	7
7	4	C
E	5	1
9	8	8
F	C	D
C	C	E
D	D	2
E	D	D
A	F	1

© 2006-2023 Jörg Striegnitz

Beispiel: Radix Sort

B Radix Sort mit D=16

0	3	A
F	C	D
1	3	3
C	0	9
3	2	F
A	F	1
9	8	8
3	4	7
D	D	2
7	4	C
E	5	1
C	C	E
E	D	D

A	F	1
E	5	1
D	D	2
1	3	3
3	4	7
9	8	8
C	0	9
0	3	A
7	4	C
F	C	D
E	D	D
C	C	E
3	2	F

C	0	9
3	2	F
1	3	3
0	3	A
3	4	7
7	4	C
E	5	1
9	8	8
F	C	D
C	C	E
D	D	2
E	D	D
A	F	1

0	3	A
1	3	3
3	2	F
3	4	7
7	4	C
9	8	8
A	F	1
C	0	9
C	C	E
D	D	2
E	5	1
E	D	D
F	C	D

© 2006-2023 Jörg Striegnitz

Eigenschaften von Radix Sort

- **In der hier vorgestellten Variante *stabil* und *nicht in situ***
  - Eigenschaften ergeben sich aus der Verwendung von Counting Sort

© 2006-2023 Jörg Striegnitz

Eigenschaften von Radix Sort

- **In der hier vorgestellten Variante *stabil* und *nicht in situ***
  - Eigenschaften ergeben sich aus der Verwendung von Counting Sort
- **Zeitkomplexität (Vergleiche)** für das Sortieren von n k-stelligen Zahlen zur Basis D

$$T(n) = \Theta\left(k \cdot (n + D)\right)$$

© 2006-2023 Jörg Striegnitz

Eigenschaften von Radix Sort

- **In der hier vorgestellten Variante *stabil* und *nicht in situ***
  - Eigenschaften ergeben sich aus der Verwendung von Counting Sort
- **Zeitkomplexität (Vergleiche)** für das Sortieren von n k-stelligen Zahlen zur Basis D

$$T(n) = \Theta\left(k \cdot (n + D)\right)$$

- **Man unterscheidet MSD- und LSD-Varianten**
  - LSD (hier vorgestellt) sortiert zuerst nach der niedrigstwertigen Ziffer
    LSD: **L**east **S**ignificant **D**igit
  - MSD sortiert zuerst nach dem höchstwertigen Ziffer. Dann ***in situ***, aber ***nicht stabil***
    MSD: **M**ost **S**ignificant **D**igit

© 2006-2023 Jörg Striegnitz

Eigenschaften von Radix Sort

- **In der hier vorgestellten Variante *stabil* und *nicht in situ***
  - Eigenschaften ergeben sich aus der Verwendung von Counting Sort
- **Zeitkomplexität (Vergleiche)** für das Sortieren von n k-stelligen Zahlen zur Basis D

$$T(n) \quad = \quad \Theta\left(k \cdot (n + D)\right)$$

- **Man unterscheidet MSD- und LSD-Varianten**
  - LSD (hier vorgestellt) sortiert zuerst nach der niedrigstwertigen Ziffer
    LSD: **L**east **S**ignificant **D**igit
  - MSD sortiert zuerst nach dem höchstwertigen Ziffer. Dann ***in situ***, aber ***nicht stabil***
    MSD: **M**ost **S**ignificant **D**igit
- **Spezialfall D=2**
  - ***Straight Radix Sort*** und ***Radix Exchange Sort***
    Werden ggf. in der Hausaufgabe behandelt

© 2006-2023 Jörg Striegnitz