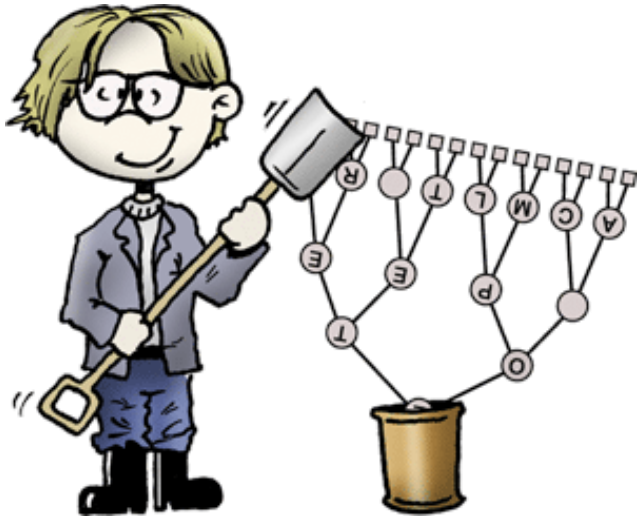
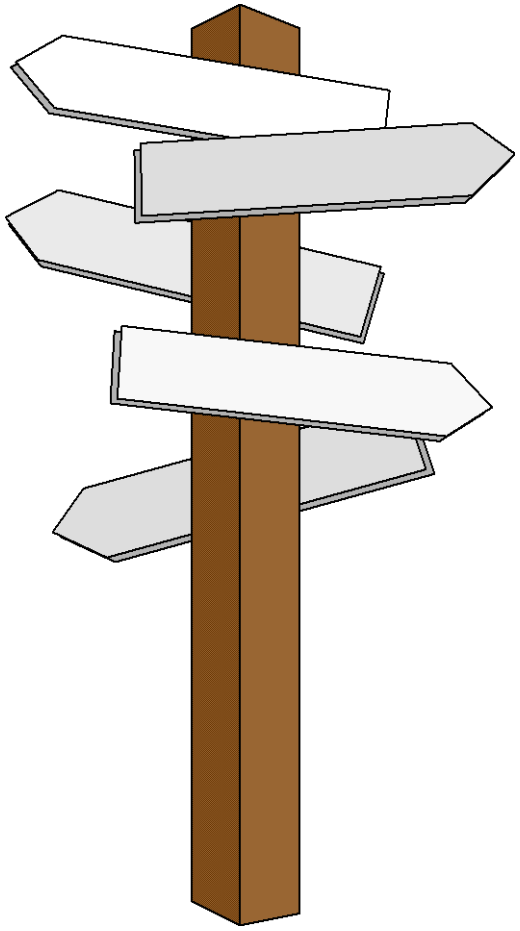
RWTH AACHEN UNIVERSITY

Algorithmen und Datenstrukturen



Vorlesung im Rahmen des dualen Studiums
„Angewandte Mathematik und Informatik“
(FH Aachen) / MATSE-Ausbildung

Prof. Dr. Hans Joachim Pflug



Kapitel 19

Sortierverfahren

RWTH AACHEN UNIVERSITY

19.1 Übersicht über die Sortierverfahren

Relevanz von Sortierverfahren

- Sortierverfahren sind Bestandteil vieler Anwendungen
- Laut Statistik: ca. 25% der kommerziell verbrauchten Rechenzeit entfällt auf Sortiervorgänge.
  (Seite 63 in T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen. 1996, Spektrum, Akademischer Verlag, Heidelberg, Berlin)
- Sortierte Datensätze können
  - viel effizienter durchsucht werden (siehe Binäre Suche)
  - leichter auf Duplikate geprüft werden
  - von Menschen leichter gelesen werden
- Auch andere praxisrelevante Aufgaben können auf Sortierproblem zurückgeführt werden, z.B.
  - Median bestimmen
  - Bestimmung der k kleinsten Elemente

Aufzählung der Sortierverfahren

- **Große Anzahl von Sortierverfahren:**

  Binary-Tree-Sort, Bogo-Sort, Bubble-Sort, Bucket-Sort, Comb-Sort, Counting-Sort, Gnome-Sort, Heap-Sort, Insertion-Sort, Intro-Sort, Merge-Sort, OET-Sort, Quick-Sort, Radix-Sort, Selection-Sort, Shaker-Sort, Shear-Sort, Shell-Sort, Simple-Sort, Slow-Sort, Smooth-Sort, Stooge-Sort

Klassifikationskriterien

- Effizienz
- Speicherverbrauch
- Intern / Extern
- Stabil / Instabil
- allgemein / spezialisiert

Klassifikation: Effizienz

- Das wichtigste Klassifikationskriterium ist die **Effizienz**.
- Einteilung in:

- Schlechter als $O(n^2)$: nicht ganz ernst gemeinte Verfahren

- $O(n^2)$: Elementare Sortierverfahren
- Zwischen $O(n^2)$ und $O(n \cdot \log n)$
- $O(n \cdot \log n)$: Höhere Sortierverfahren

- Besser als $O(n \cdot \log n)$: Spezialisierte Verfahren.

Klassifikation: Effizienz (2)

- Von diesen Verfahren sollten Sie die Effizienz kennen:

- $O(n^2)$: Elementare (Einfache) Sortierverfahren
  - **Bubble-Sort, Insertion-Sort, Selection-Sort**
- Zwischen $O(n^2)$ und $O(n \cdot \log n)$
  - **Shell-Sort**
- $O(n \cdot \log n)$: Höhere Sortierverfahren
  - **Heap-Sort, Merge-Sort, Quick-Sort**
- Besser als $O(n \cdot \log n)$: Spezialisierte Verfahren.
  - **Radix-Sort**

Klassifikation: Effizienz (Spezialfälle)

- Spezialfälle sind:
  - **Datenmenge sehr klein (<50)**
  - **Datenmenge sehr groß (passt nicht in Hauptspeicher)**
  - **Daten sind schon „vorsortiert“ (nur wenige Elemente sind nicht am richtigen Platz).**
  - **Daten stehen nicht in einem Feld sondern in einer verketteten Liste.**

Weitere Klassifikationskriterien

- Speicherverbrauch
- Rein vergleichsbasiertes Verfahren (sehr universell einsetzbar) oder spezialisiertes Verfahren (muss an jeweiliges Problem angepasst werden):

- Stabil / Instabil

  (behalten Datensätze mit gleichen Schlüsseln relative Reihenfolge?)
  Relevant bei mehrmaligem Sortieren nach verschiedenen Schlüsseln

Beispiel für stabiles Sortieren

- Namen nach Vor- und Nachname sortieren:
  „Meier, Martin“, „Meier, Ulla“, „Schmitz, Heinz“, „Ernst, Eva“

1. Sortieren zuerst nach Vornamen:
   „Ernst, Eva“, „Schmitz, Heinz“, „Meier, Martin“, „Meier, Ulla“

2. Dann Sortieren nach Nachnamen:
   - Stabil: „Ernst, Eva“, „Meier, Martin“, „Meier, Ulla“, „Schmitz, Heinz“
   - Instabil, z.B. „Ernst, Eva“, „Meier, Ulla“, „Meier, Martin“, „Schmitz, Heinz“

- Das wichtigste stabile Verfahren ist Merge-Sort. Außerdem sind Radix-Sort, Insertion-Sort und Bubble-Sort stabil.

Internes / Externes Sortieren

- Voraussetzung für bisher behandelte Verfahren:
  Schneller Zugriff auf einen beliebigen Datensatz (wahlfreiem Zugriff).
  - Bezeichnung „Internes Sortieren“

- Dies ist in manchen Fällen nicht möglich:
  - bei sehr großen Datenbeständen z.B. auf Hintergundspeichern (externen Speichern) mit sequentiellem Zugriff.
  - bei verketteten Listen.

- Hier werden Verfahren verwendet, die lediglich sequentiellen Zugriff benötigen.
  - Bezeichnung „Externes Sortieren“.
  - Wichtigstes Verfahren: Merge-Sort.

Beste Sortierverfahren

- Normalerweise **Quicksort.**

- Merge-Sort, falls
  - **die Datenmenge zu groß für den Hauptspeicher ist.**
  - **die Daten als verkettete Liste vorliegen.**
  - **ein stabiles Verfahren nötig ist.**

- Insertion-Sort, falls
  - **wenige Elemente zu sortieren sind.**
  - **die Daten schon vorsortiert sind.**

- Radix-Sort, falls
  - **sich ein hoher Programmieraufwand für ein sehr schnelles Verfahren lohnt.**

Hilfsmethoden zum Sortieren (Java)

```
static void swap(int array[], int index1, int index2) {
      int temp;
      temp = array[index1];
      array[index1] = array[index2];
      array[index2] = temp;
}
```

Für Klassen, die das Interface List implementieren (ArrayList, …) geht auch:

```
Collections.swap(List<?> list, int i, int j)
```

19.2 Nicht ganz ernstzunehmende Verfahren

O(n·n!): Bogo-Sort

- „The archetypal perversly awful algorithm“ (Wikipedia).
- Würfelt solange alle Elemente durcheinander, bis das Feld (zufällig) sortiert ist.
- Zitat aus dem Internet:
  - **Looking at a program and seeing a dumb algorithm, one might say „Oh, I see, this program uses bogo-sort.“**

```
public void sort(List a) {
  while (isSorted(a)==false) {
    Collections.shuffle(a);
  }
}
```

```
public boolean isSorted(List a) {
  for (int i=1; i<a.size(); i++) {
    if isLastSmaller(a.get(i-1),
                     a.get(i)) {
      return false;
    }
  }
  return true;
}
```

$O(n^{\log(n)/2})$: Slow-Sort

- Vorgehensweise wird hier nicht erklärt
  - **-> Wikipedia**
- Versucht, die Arbeit soweit wie möglich zu vervielfachen (multiply and surrender).
- Das Ergebnis wird erst dann berechnet, wenn die Lösung nicht weiter hinausgezögert werden kann.
- Ziel: Auch im besten Fall ineffizienter als alle anderen Sortierverfahren.
- Ist aber trotzdem ein echtes Sortierverfahren.

O(n³): Stupid-Sort

- Sehr kurzer Code.
- Variante von Bubble-Sort.
- Kann auch rekursiv implementiert werden („*with recursion, stupid-sort can be made even more stupid*“ (Wikipedia)).

```
public void sort(int[] a) {
  for (int i=0; i<a.length-1; i++) {
    if (a[i]>a[i+1]) {
      swap(a, i, i+1);
      i=-1;
    }
  }
}
```

19.3 O(n²): Einfache Sortierverfahren

Übersicht

- Die bekannten einfachen Sortierverfahren sind:

Bubble-Sort **Selection-Sort** **Insertion-Sort**

- Das beste Verfahren der 3 ist **Insertion-Sort**. Es wird ausführlich vorgestellt.
- Selection-Sort wird kurz angesprochen.
- Bubble-Sort wird nicht weiter erklärt.

- Dafür wird ein besonders einfacher Algorithmus namens **Simple-Sort** behandelt.

Simple-Sort / Selection-Sort

- Simple-Sort
  - **$O(n^2)$, mit hohem Vorfaktor.**
  - **Leicht zu merken.**
- Wenden Sie es an:
  - **wenn Sie keine Zeit oder Lust zum Nachdenken haben.**
  - **wenn die Felder so klein sind, dass der Algorithmus nicht effektiv sein muss.**
  - **wenn niemand sonst Ihren Code zu sehen bekommt.**
- Selection-Sort
  - **Eines der wichtigeren elementaren Verfahren.**
  - **Wird ein elementares Verfahren benötigt, wird aber meistens das etwas schnellere InsertionSort verwendet.**

Prinzip: Simple-Sort und Selection-Sort

- Das Grundprinzip ist für beide Sortierverfahren gleich
- SimpleSort ergibt einen besonders einfachen Code, ist aber langsamer.
- Grundprinzip:

for (int i=0; i<array.length-1; i++)
Suche das kleinste Element zwischen i und dem rechten Feldende.
Vertausche dieses Element mit dem Element i.

Simple-Sort (Code)

- Das Suchen und Vertauschen wird bei SimpleSort auf ganz spezielle Weise gemacht:
  - **Gehe vom i. Element aus nach rechts.**
  - **Jedes Mal, wenn ein kleineres Element als das auf Position i auftaucht, dann vertausche es mit dem i. Element.**

```
public void simpleSort (int[] a) {

  for (int i=0; i<a.length; i++) {
    for (int j=i+1; j<a.length; j++) {
      if (a[i]>a[j]) {
        swap (a,i,j);
      }
    }
  }
}
```

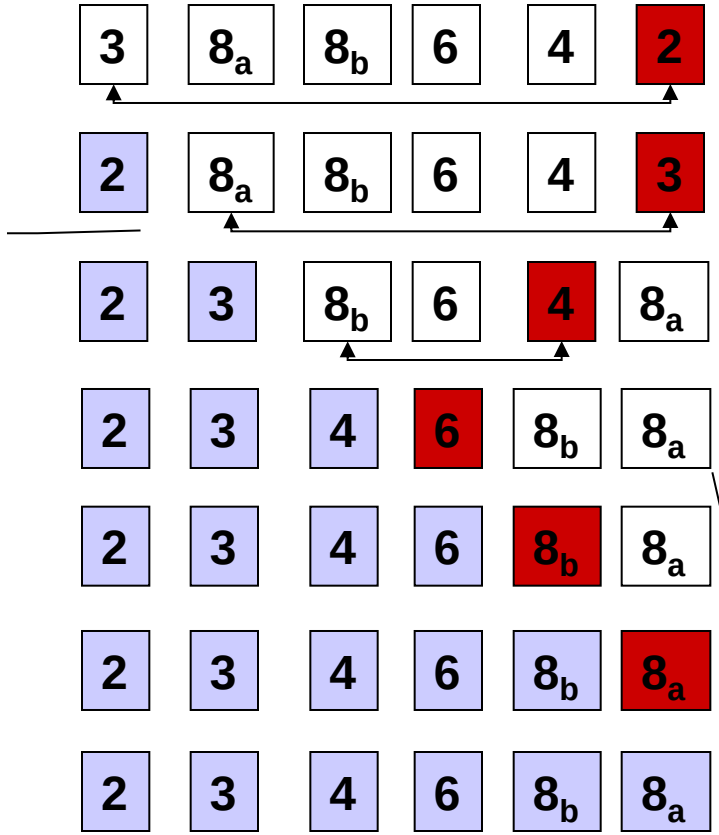
Selection-Sort (Code)

- „Normale“ Vorgehensweise: Erst kleinstes Element suchen, dann mit Element i vertauschen.

```
public void selectionSort (int[] a) {

  for (int i=0; i<a.length; i++) {
    int small = i;
    for (int j=i+1; j<a.length; j++) {
      if (a[small]>a[j]) {
        small = j;
      }
    }
    swap(a, i, small);
  }
}
```

Beispiel zu Selection-Sort



SelectionSort ist instabil:

Sortierte Teilfolge

Kleinstes Element

Keine Vorteile, wenn Feld schon sortiert.

Sortieren durch Einfügen: InsertionSort

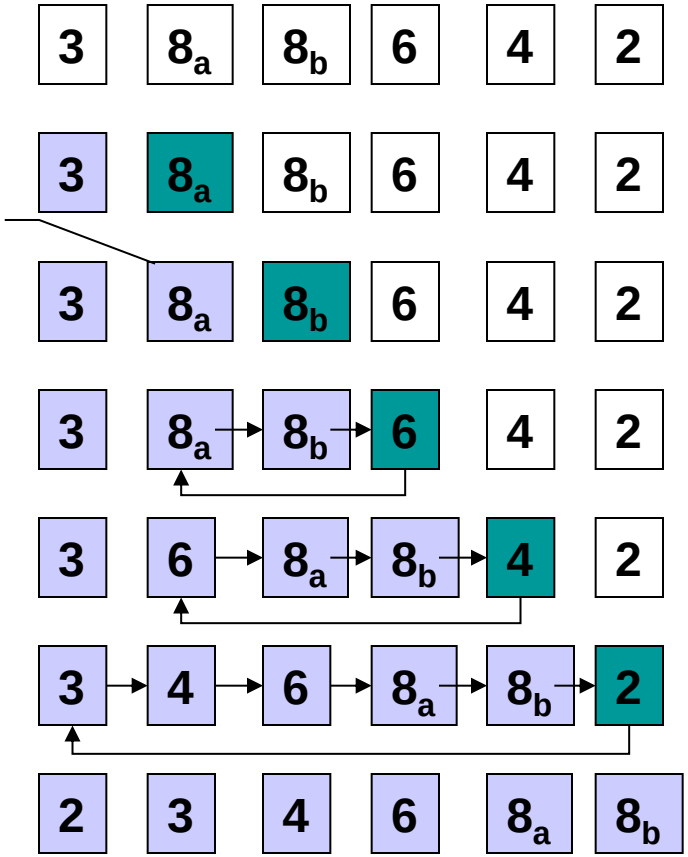
- In den meisten Fällen der schnellste elementare Suchalgorithmus.

Grundidee (am Beispiel der Sortierung eines Kartenstapels)

1. **Starte mit der ersten Karte einen neuen Stapel.**
2. **Nimm jeweils die nächste Karte des Originalstapels und füge diesen an der richtigen Stelle in den neuen Stapel ein.**

Beispiel zu InsertionSort

InsertionSort ist stabil: Vertauschung nur bei Ungleichheit



Sortierte Teilfolge

Einzusortierendes Element

InsertionSort in Java

```
public static void insertionSort(int[] array){

    for (int i=1; i < array.length; i++) {
        int m = array[i];

        // fuer alle Elemente links von aktuellem Element
        int j;
        for (j=i; j>0; j--) {
            if (array[j-1]<=m) {
                break;
            }
            // größere Elemente nach hinten schieben
            array[j] = array[j-1];
        }

        // m an freiem Platz einfügen
        array[j] = m;
    }
}
```

Komplexitätsanalyse von InsertionSort

- Wir zählen die Anzahl der Vergleiche (Anzahl der „Bewegungen“ ist ungefähr gleich).

- ***n-1* Durchläufe**: Im Durchlauf mit Nummer *k* (*k* $\in$ *[2;n]*):
  - höchstens *k-1* Vergleiche.
  - mindestens 1 Vergleich (und keine Bewegung).

- **Best Case** - vollständig **sortierte** Folge: *n-1* Vergleiche, keine Bewegungen ➔ O(n).

- **Average Case** - jedes Element wandert **etwa in Mitte des sortierten Teils**:

$$\sum_{k=2}^{n} k/2 = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) \approx \frac{n^2}{4}$$ Vergleiche ➔ O(n^2).

Bewertung

SimpleSort:

- Einfach zu implementieren.
- Langsam.

SelectionSort:

- Aufwand ist unabhängig von der Eingangsverteilung (Vorsortierung).
- Es werden nie mehr als O(n) Vertauschungen benötigt.

BubbleSort:

- Stabil
- Vorsortierung wird ausgenutzt.
- Langsam

InsertionSort:

- Stabil
- Vorsortierung wird ausgenutzt
- Für ein elementares Suchverfahren ($O(n^2)$) schnell.

Analyse einfacher Sortierverfahren

Verfahren	Laufzeitmessungen (nach Wirth, Sedgewick)	Vosortierung ausnutzen	Stabil
SimpleSort	*330**		
SelectionSort	*120-200*		
InsertionSort	*100*	*X*	*X*
BubbleSort	*250-400*	*X*	*X*

* Eigene Messung

InsertionSort:

- Stabil
- Vorsortierung wird ausgenutzt
- Für ein elementares Suchverfahren ($O(n^2)$) schnell.

O(n log n): Schnelle Sortiervahren

19.4 Quick-Sort

Übersicht

- Die bekannten höheren Sortierverfahren sind:

Quick-Sort	**Merge-Sort**	**Heap-Sort**

- Alle 3 Verfahren werden vorgestellt.

Quick-Sort: Verbreitung

- Die überwiegende Mehrheit der Programmbibliotheken benutzt Quick-Sort.
  - **z.B. Java, C#, …**
  - **In fast allen Fällen sind zwei Optimierungen eingebaut: „Median of three“ und „Behandlung kleiner Teilfelder“.**
- Gnu-C++ benutzt Intro-Sort (Quick-Sort-Variante).

- Bei Objekten kann Stabilität wichtig sein. Hier verwendet Java (wie Python) Merge-Sort.

QuickSort: Prinzip

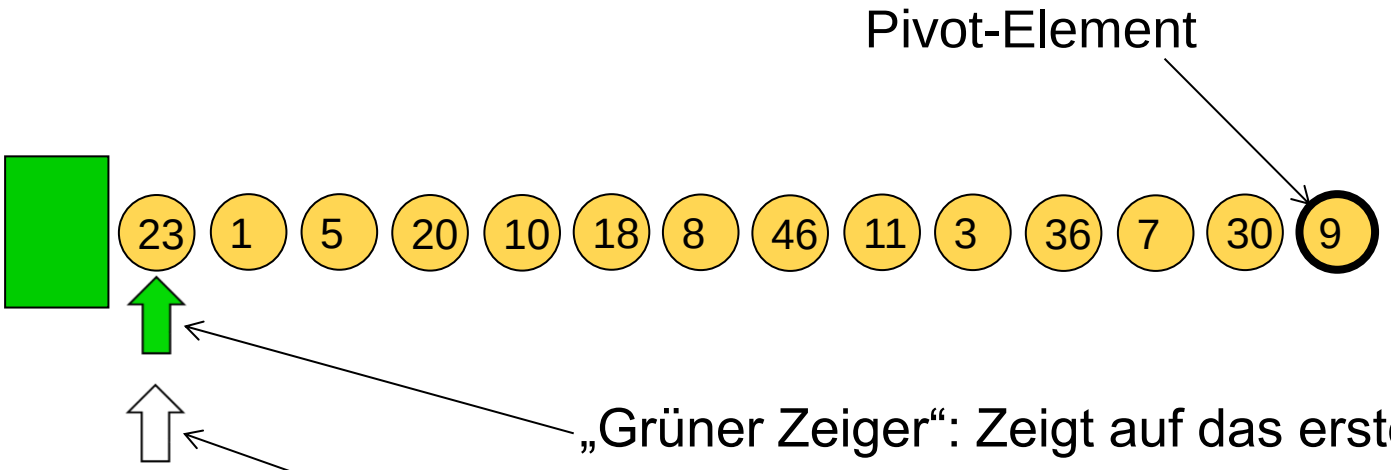
Prinzip: *divide-and-conquer* (‚teile' und ‚herrsche')

Rekursiver Algorithmus

- Müssen 0 oder 1 Elemente sortiert werden $\Rightarrow$ Rekursionsabbruch
- Wähle ein Element als „Pivot-Element" aus.
- Teile das Feld in 2 Teile:
  - **Ein Teil mit den Elementen größer als das Pivot.**
  - **Ein Teil mit den Elementen kleiner als das Pivot.**
- Wende den Algorithmus rekursiv für beide Teilfelder an.

Quicksort: Beispiel (1)

- Start

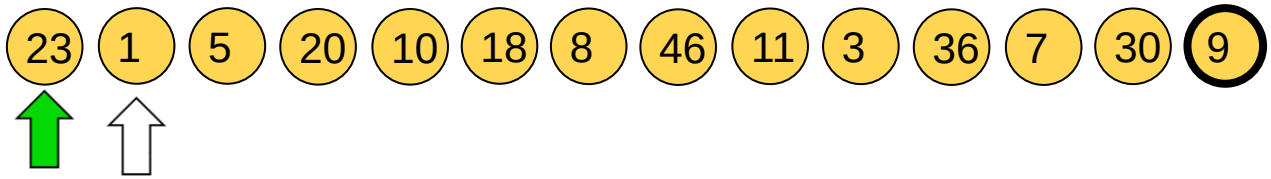


„Grüner Bereich“: Alle Element hier sind ≤ Pivot.

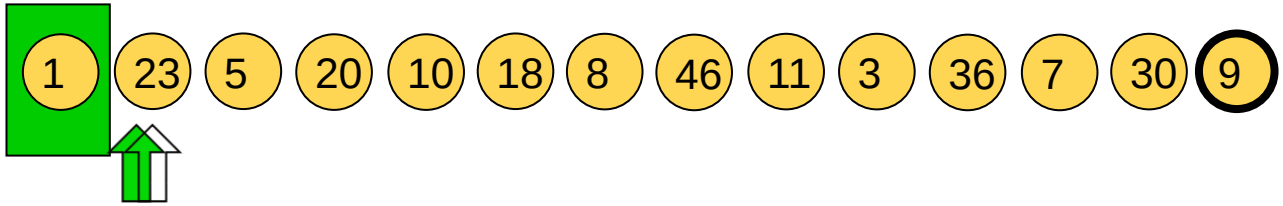
Quicksort: Beispiel (2)

- Erster Schritt

Der Suchzeiger rückt nach rechts und sucht ein Element ≤ Pivot.

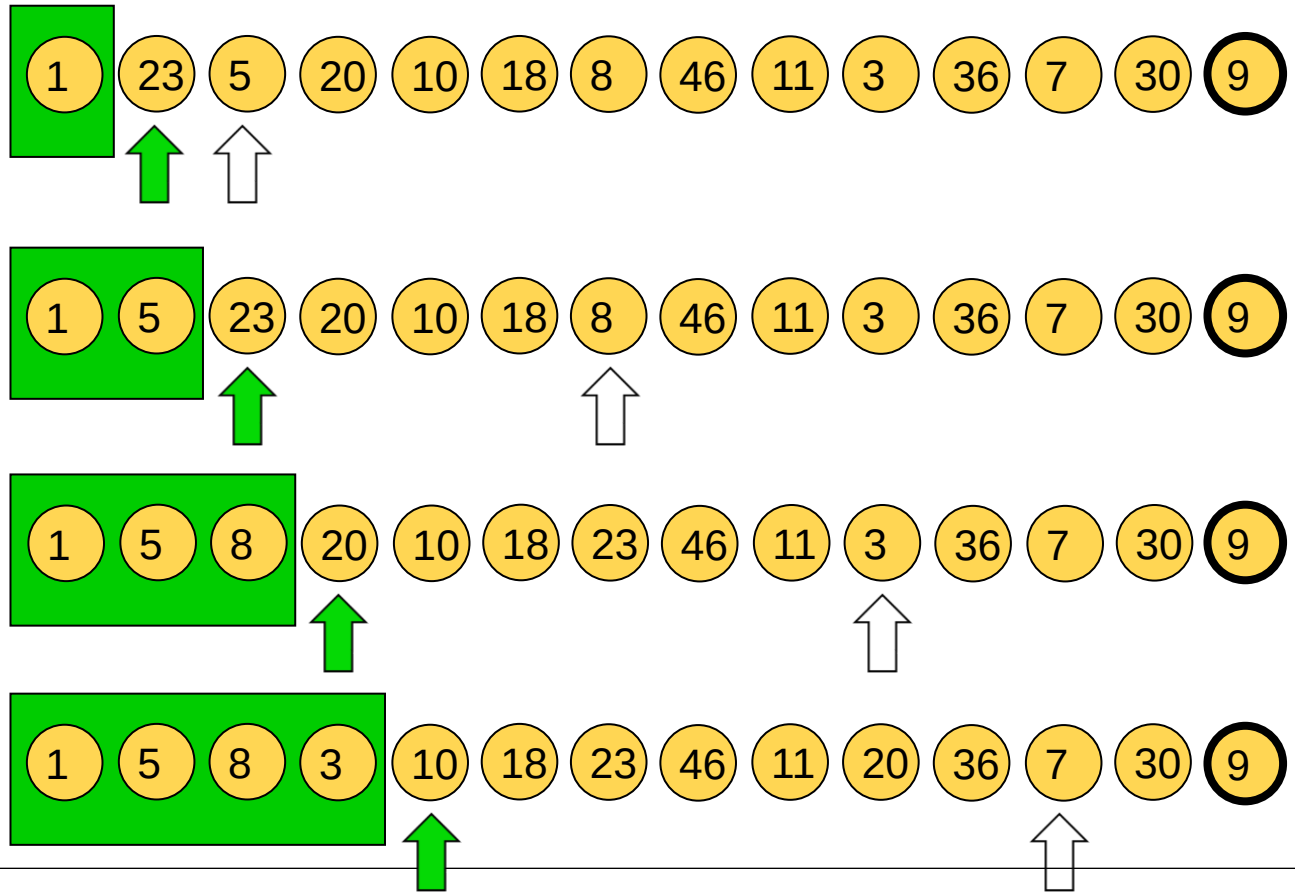


Die Elemente unter dem grünen und dem Suchzeiger werden getauscht.
Der grüne Bereich wird um 1 Element nach rechts erweitert.

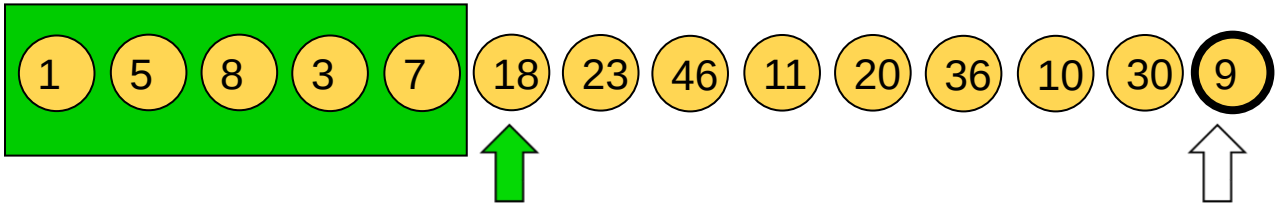


Quicksort: Beispiel (3)

Dieser Schritt wird solange wiederholt, bis der Suchzeiger am rechten Rand des Feldes angekommen ist.

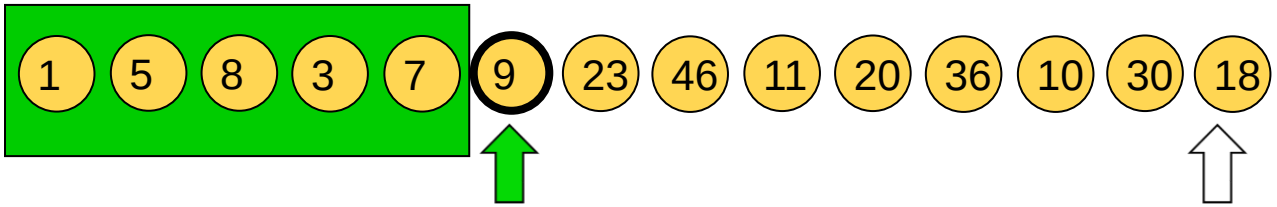


Quicksort: Beispiel (4)



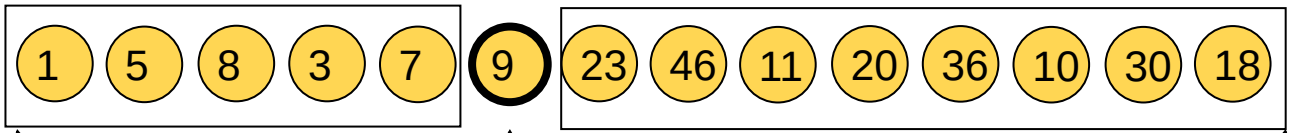
Jetzt sind alle Elemente ≤ Pivot (außer dem Pivot-Element selbst) im grünen Bereich.

Zum Abschluss wird das Pivot-Element mit dem ersten Element hinter dem grünen Bereich vertauscht.



Quicksort: Beispiel (5)

Es gibt jetzt 3 Bereiche:



Bereich 2: Das Pivot-Element befindet sich bereits an der richtigen Stelle.

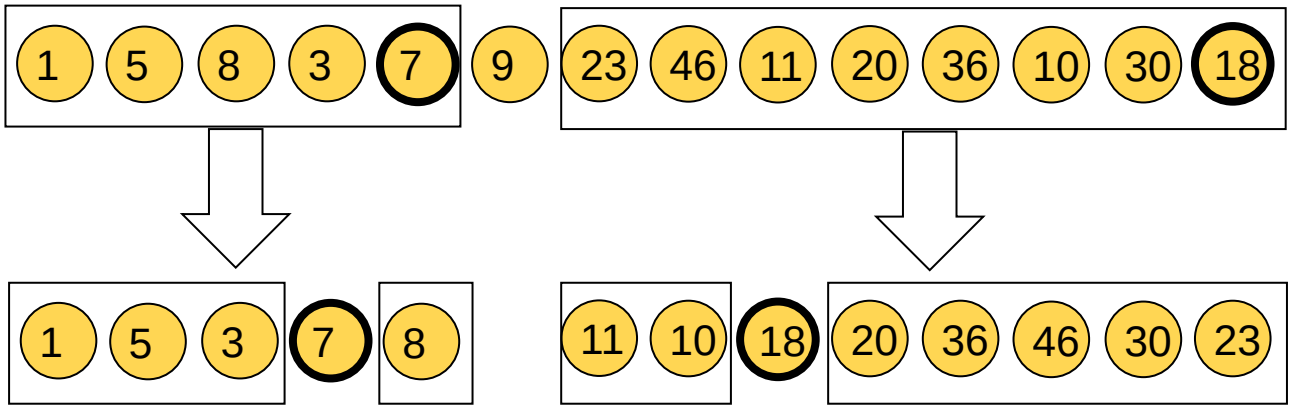
Bereich 1 (vor dem Pivot-Element): Alle Wert sind ≤ Pivot.

Bereich 3 (hinter dem Pivot-Element): Alle Wert sind > Pivot.

Die Bereiche 1 und 3 werden rekursiv mit Quicksort sortiert.

Quicksort: Beispiel (5)

Die Bereiche 1 und 3 werden rekursiv mit Quicksort sortiert.



Die Rekursionen setzen sich solange fort, bis die Teilbereiche nur noch 1 Element enthalten (siehe auch Folie „Optimierung: Rekursionsabbruch“).

Zeitkomplexität von Quick-Sort: Best Case

Pivot-Wert ist immer Median der Teilliste ⇒ Teillisten werden stets halbiert.

Stufe 1

1 n

n Elemente werden mit dem Pivotwert verglichen und maximal n/2 Paare vertauscht.

Stufe 2

1 n/2 n

In jeder der beiden Teillisten werden n/2 Elemente (also insgesamt n) mit dem jeweiligen Pivotwert verglichen und maximal n/4 Paare vertauscht.

Allgemein: In jeder Stufe werden n Elemente betrachtet. Abbruch bei Teillisten der Länge $n \leq 1 \Rightarrow \lfloor \mathrm{ld}\, n \rfloor$ Stufen (mit Halbierung der Teillisten).

Also: $T^b_{\mathrm{quicksort}}(n) \in \boldsymbol{O(n \log n)}$

Zeitkomplexität v. Quick-Sort: Worst Case

Als Pivot-Wert wird stets das größte oder kleinste Element der Teilliste ausgewählt.

Dann gilt:

- Die Länge der längsten Teilliste ist (n-1) bei Stufe 1,
  (n-2) bei Stufe 2, etc.
  Allgemein : (n-i) bei Stufe i.
- Es sind (n-1) Stufen nötig.
  In jeder Stufe i werden n-i Elemente betrachtet.
- Also: $T^{w}_{quicksort}(n) \in \boldsymbol{O(n^2)}$

Zeitkomplexität v. Quick-Sort: Average Case

Aufwand im Mittel:

Genaue Analyse ist aufwändig. Resultat: $T^{av}_{quicksort}(n) \in \mathbf{O(n \log n)}$

Genauer:
#Vergleiche im mittleren Fall ist nur um etwa 39% größer als im Besten Fall. QuickSort ist also auch im Mittleren Fall eine sehr gute Wahl.

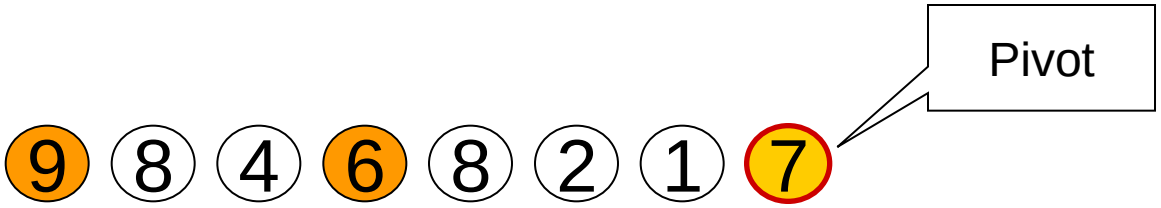
Hauptziel ist Vermeidung des schlimmsten Falls.

Probleme von Quick-Sort: Pivot-Elemente

- Wenn die Pivot-Elemente unglücklich gewählt sind, erhält man das wesentlich schlechtere Zeitverhalten $O(n^2)$.
- Dazu Anmerkung:
  Wenn man beim Zerlegen das erste oder letzte Feldelement als Pivot wählt, dann liegt der Worst Case bei sortierten Feldern vor.
  - **Auch nahezu sortierte Felder haben schon $O(n^2)$.**
- Pivot geschickter wählen.
  - **Einfachste Möglichkeit: Mittleres Element als Pivot wählen. Dann liegt der Best case bei sortierten Felder vor.**
  - **Man will den Quick-Sort aber noch sicherer machen.**

Optimierung: Pivot-Elemente

- *Median-of-three-Methode* zum Auswählen des Pivots:
  - Es werden drei Elemente als Referenz-Elemente, z.B. vom Listenanfang, vom Listenende und aus der Mitte gewählt. Das Element mit dem mittleren Schlüsselwert wird als Pivot-Element gewählt.
  - Kann auf mehr als drei Elemente ausgebaut werden.
  - Das Pivot-Element wird vor dem Quicksort-Durchgang mit dem letzten Element getauscht.



Probleme von Quick-Sort: Rekursionsabbruch

- Einfache Lösung: Rekursionsabbruch, wenn die Teilliste 0 oder 1 Element enthält.
- Aber: Die letzten Rekursionsdurchgänge sind nicht mehr effektiv.

- Daher wird die Rekursion schon früher abgebrochen.

Optimierung: Rekursionsabbruch

- **Kleine Teilliste mit InsertionSort sortieren**:
  - Die Grenze für eine kleine Teilliste ist nicht klar festgelegt.
    - Die Standardwerke (Knuth, Sedgewick) empfehlen 9.
    - Im Internet findet man aber auch andere Werte zwischen 3 und 32.

Sprache	Bibl.-Aufruf	Rekursionsabbruch bei max. n Elementen
Java (Sun)	java.util.Arrays.sort	6
Java (Gnu)	java.util.Arrays.sort	7
C# / Mono	System.Collections.ArrayList.sort	3
Ruby	Array.sort	2

Bewertung

Quicksort:

- Schnellster der oft verwendeten allgemeinen Algorithmen.
- Programmbibliotheken implementieren ganz überwiegend QuickSort.
- Nutzt Vorsortierung nicht aus.
- Im Worst Case nur $O(n^2)$.
- Schlecht für kleine n (<20).
- Zwei Standard-Optimierungen

Standard-Optimierung: Rekursionsabbruch für kleine n:

- Verwendung von InsertionSort
- Grenze liegt zwischen 3 und 32.

Standard-Optimierung: Pivot ist Median mehrerer Elemente:

- 3 oder mehr Pivot-Elemente.
- Dadurch Worst Case sehr unwahrscheinlich.
- Langsamer aufgrund zusätzlicher Rechnenschritte.

Optimierte Variante: Intro-Sort

- Optimierte Quicksort-Variante
  - **Z.B. in Gnu C++**
- Ab einer gewissen Rekursionstiefe wird zu Heap-Sort gewechselt.
  - **Heap-Sort hat O(n log n) auch im schlechtesten Fall. Damit kann $O(n^2)$ nicht auftreten.**
  - **Allgemein ist Heap-Sort aber langsamer als Quick-Sort.**
  - **Große Rekursionstiefe deutet bei Quick-Sort auf Worst-Case-Probleme. Daher wechselt man hier zu Heap-Sort.**

Optimierte Variante: Dual-Pivot Quicksort

- Yaroslavskiy, Bentley, Bloch (2009)
- Java benutzt Double-Pivot Quicksort seit **Java 7**

- 2 Pivot-Elemente
- Werte werden in 3 Bereiche geteilt

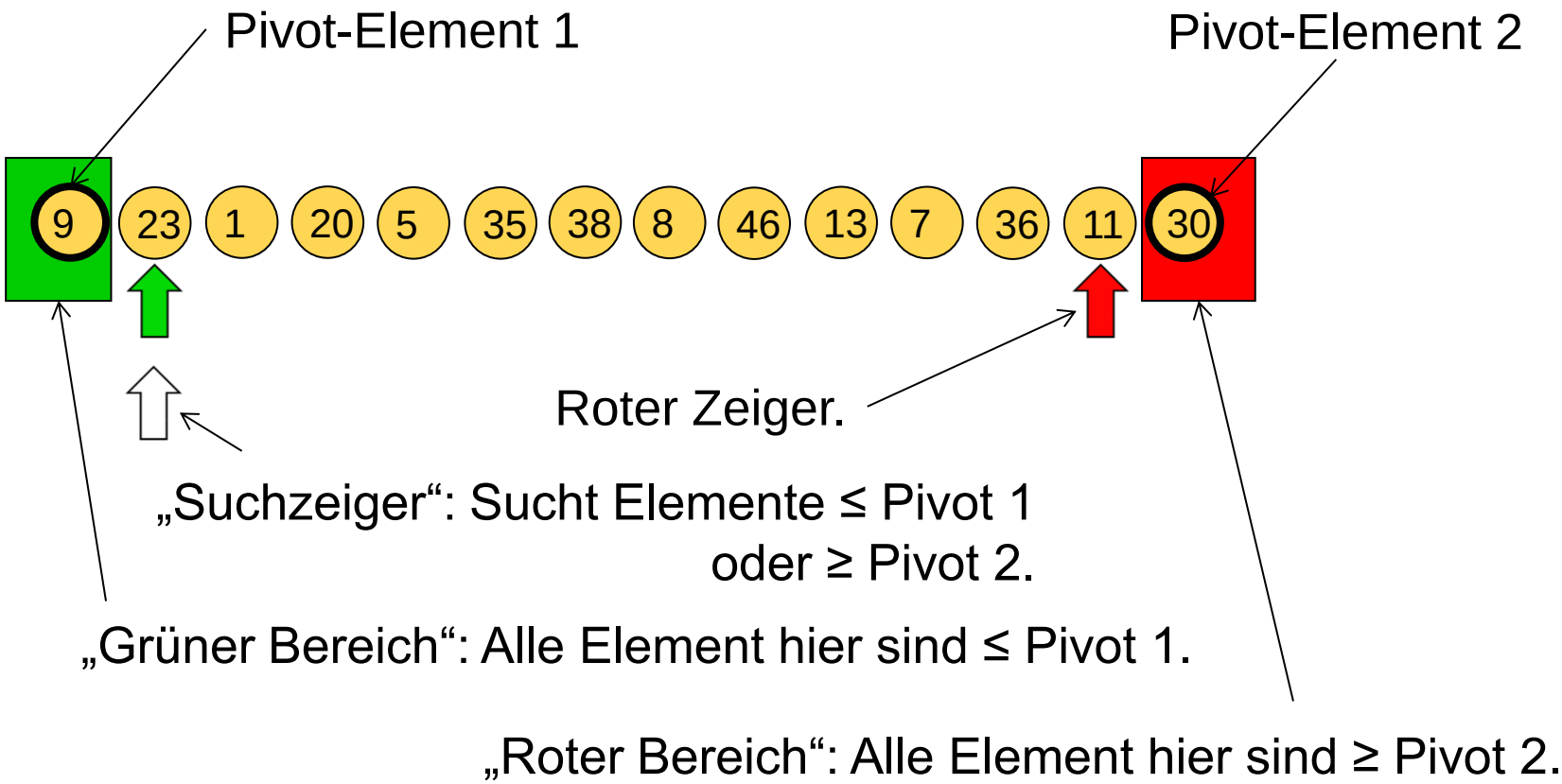
Dual-Pivot Quicksort

- Es gibt Ähnlichkeiten zur vorgestellten Quick-Sort-Variante.

- Zunächst werden zwei Pivot-Elemente ausgewählt.
- Das größere Element wird ganz nach rechts kopiert.
- Das kleinere Element wird ganz nach links kopiert.

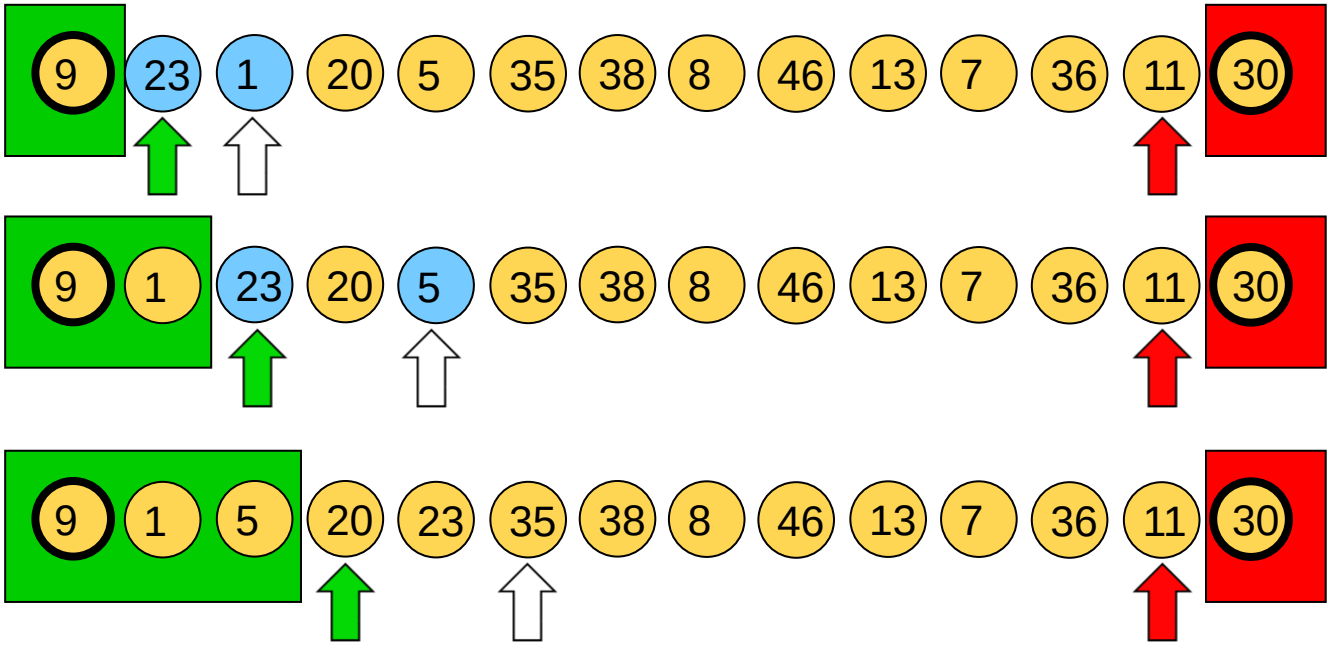
Dual-Pivot-Quicksort: Beispiel (1)

- Start



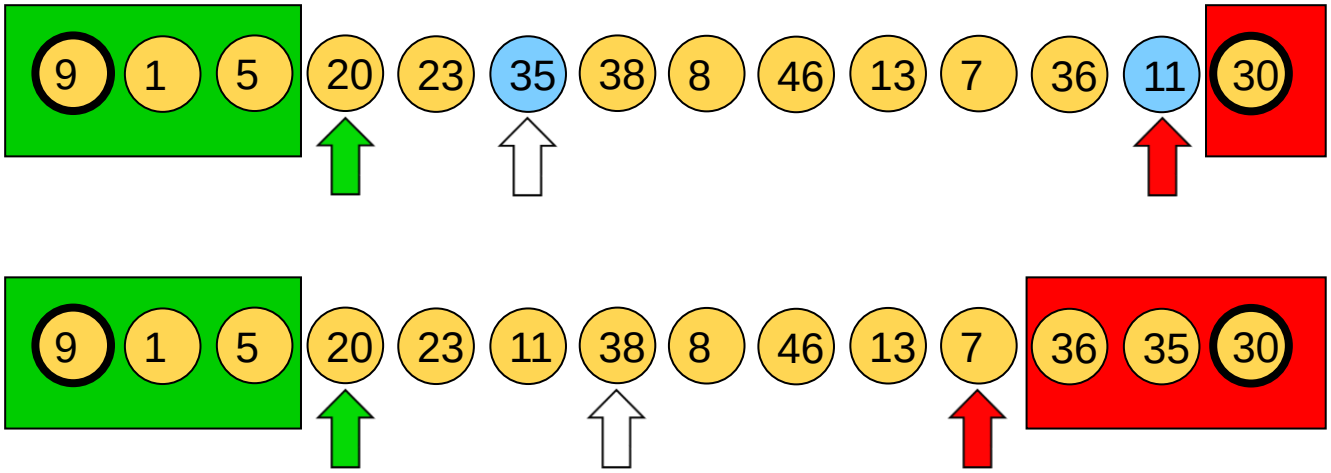
Dual-Pivot-Quicksort: Beispiel (2)

- Trifft der Suchzeiger auf Elemente ≤ Pivot 1 ist das Verfahren genau so wie bei der vorgestellten „normalen“ Quick-Sort-Variante.



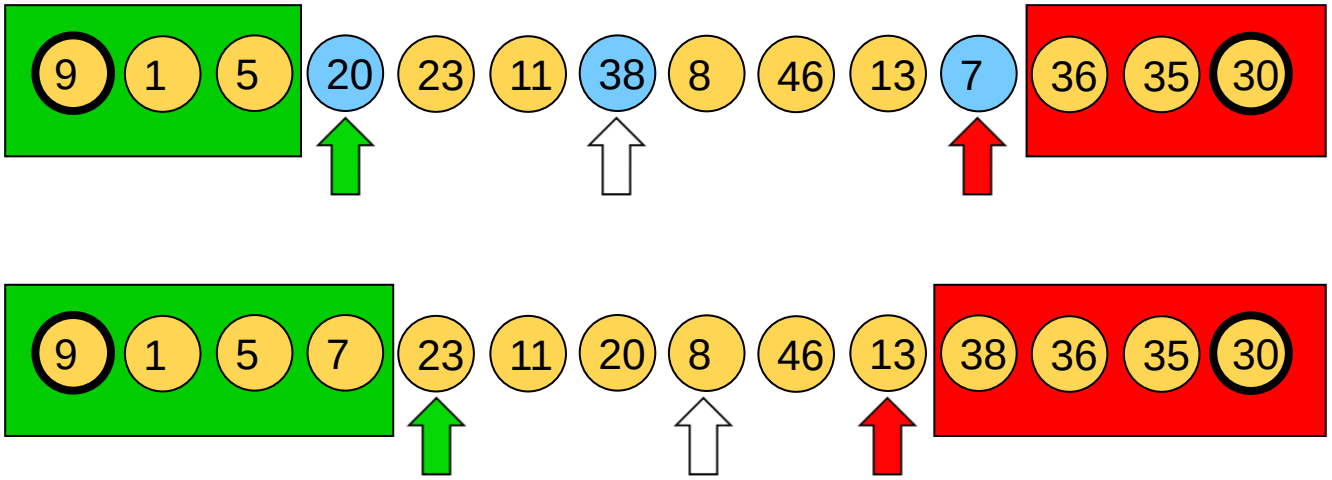
Dual-Pivot-Quicksort: Beispiel (3)

- Trifft der Suchzeiger auf Elemente ≥ Pivot2, dann vertauschen die Elemente unter dem Suchzeiger und dem roten Zeiger. Der rote Zeiger rückt zunächst um 1 Element vor.
- Besonderheit: Solange der rote Zeiger jetzt auf Elementen ≥ Pivot2 steht, rückt er weiter nach links vor.
- Das zurückgetauschte Element (11) gehört nicht in den grünen Bereich und kann stehenbleiben.



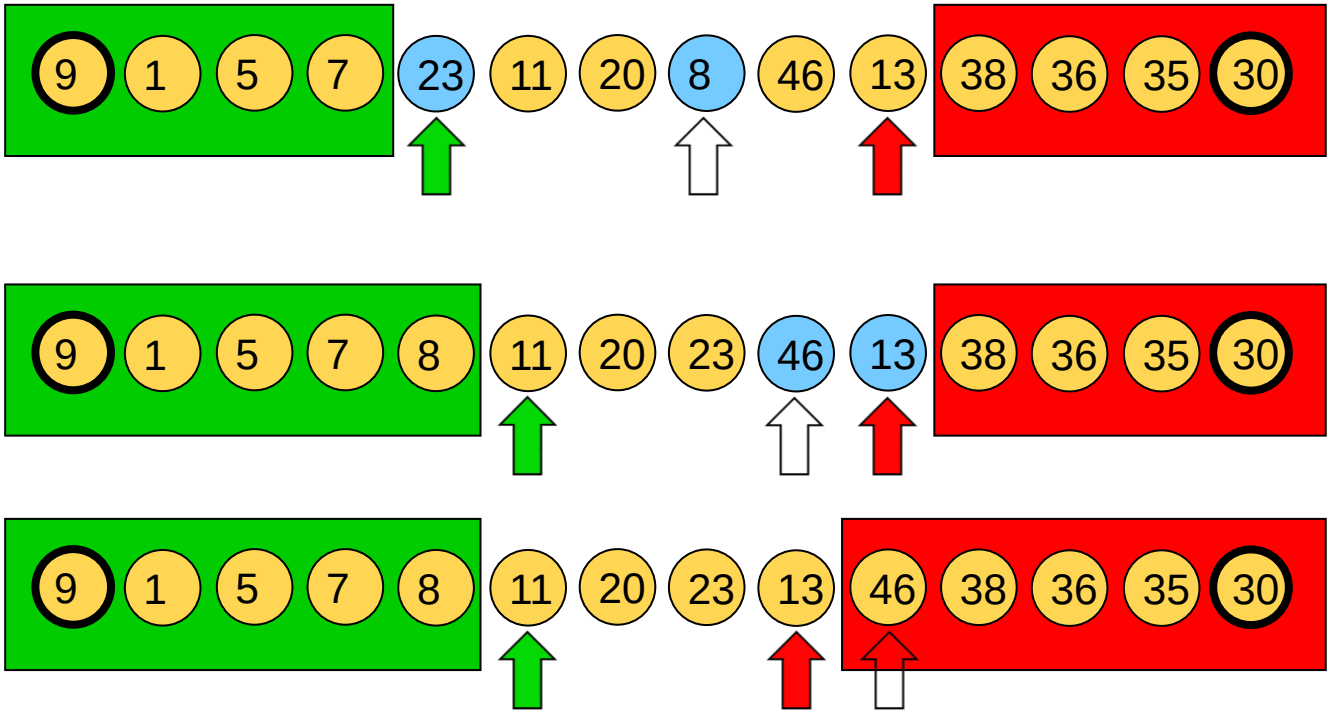
Dual-Pivot-Quicksort: Beispiel (4)

- Die 38 vertauscht mit der 7. Da die 7 in den grünen Bereich gehört, vertauscht sie anschließend noch mit der 20.
- Sowohl der rote als auch der grüne Zeiger rücken danach eine Position nach vorne.



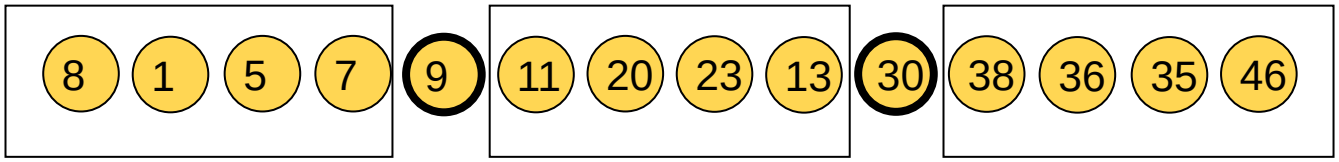
Dual-Pivot-Quicksort: Beispiel (5)

- Sobald der Suchzeiger in den roten Bereich wandert, kann man die Vertauschungen beenden.



Dual-Pivot-Quicksort: Beispiel (6)

- Man kopiert die beiden Pivots an den inneren Rand ihres Bereichs. Sie haben dort bereits ihren endgültigen Platz. Die restlichen drei Bereiche sortiert man rekursiv mit Quicksort weiter.



O(n log n): Schnelle Sortiervahren

19.5 Heap-Sort

ADT Prioritätswarteschlange

- Auch **Vorrangwarteschlange** oder **Priority Queue** genannt.
- Eine Warteschlange, deren Elemente einen Schlüssel (Priorität) besitzen.

- Wichtige Operationen bei Prioritätswarteschlangen:
  - Element in Schlange einfügen
  - Element mit der höchsten Priorität entnehmen.
    - Dies ist gewöhnlich das Element mit dem kleinsten Schlüssel,
    - Manchmal ist es auch das Element mit dem größten Schlüssel.

Anwendungen

- Ereignissimulation
  - **Die Schlüssel sind die Zeitpunkte von Ereignissen, die in chronologischer Reihenfolge zu verarbeiten sind.**
- Verteilung der Rechenzeit auf mehrere Prozesse
- Graphalgorithmen
  - **Dijkstra, Vorlesung 15**
- Sortierverfahren
  - **Alle Elemente in Prioritätswarteschlange einfügen**
  - **Nach der Reihe die größten Elemente entnehmen**
  - **Heap-Sort**

Anwendung außerhalb des EDV-Bereichs

- Wartezimmer eines Arztes
- Reihenfolge des Aufrufs wird durch Prioritätswarteschlange bestimmt.
- Die Priorität wird ermittelt aufgrund:
  - **Ankunftszeit**
  - **Termin / kein Termin**
  - **Privatpatient / Kassenpatient**
  - **Notfall / kein Notfall**

Datenstrukturen für Pr.-Warteschlange

- Einfache Implementierung mit Array (Feld).
- Effiziente Implementierung mit AVL-Baum.
- Andere effiziente Implementierung: Partiell geordneter Baum (**Heap**).

Prioritätswarteschlangen in Java

- Kein Interface
- Nur eine einzige Implementation einer Prioritätswarteschlange: **`java.util.PriorityQueue`**
  - (wird auch in `java.util.concurrent.PriorityBlockingQueue` benutzt).

- Die zugrundeliegende Datenstruktur ist ein Heap.

Heap

- Das Wort Heap (Halde) hat zwei Bedeutungen:

1. **Heap:** Besonderer Speicherbereich, in dem Objekte und Klassen gespeichert werden.
2. **Heap:** Datenstruktur zur effizienten Implementierung einer Prioritätswarteschlange.

- Beide Bedeutungen haben nichts miteinander zu tun. In folgenden Kapitel widmen wir uns ausschließlich der zweiten Bedeutung.
- Dabei betrachten wir ausschließlich den **binären Heap**. Es gibt z.B. noch den Binominial-Heap und den Fibonacci-Heap.

Einführung:

http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

Kapitel Heap (1+2)

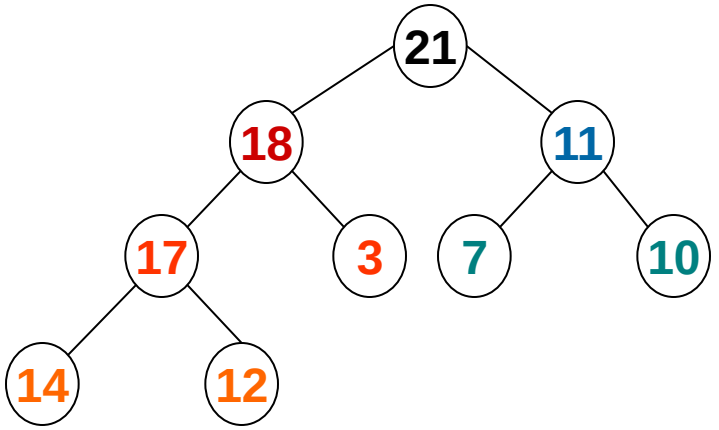
Definition:

Ein ***Heap*** ist ein Binärbaum mit folgenden Eigenschaften:

- Er ist links-vollständig
- Die Kinder eines Knotens sind höchstens so groß wie der Knoten selbst.

⇒ das größte Element befindet sich an der Wurzel des Heaps

Achtung: In der Literatur gibt es auch die umgekehrte Definition



Heap-Eigenschaften

Ein Heap ist also ein Binärbaum mit den beiden Eigenschaften:

Form :

(linksvollständig)

und

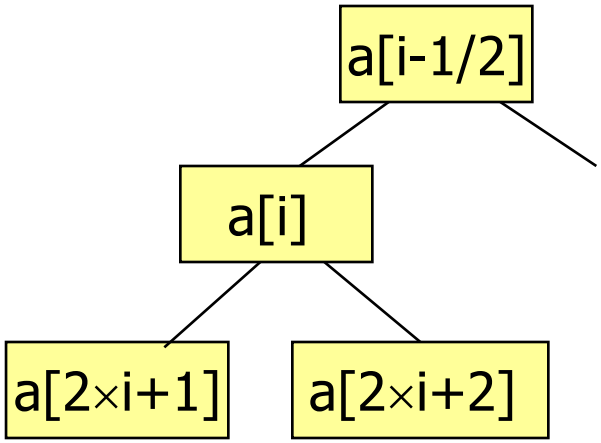
Ordnung:

Entlang jedes Pfades von einem Knoten zur Wurzel sind die Knoteninhalte aufsteigend sortiert.

Feld-Einbettung

Vorteil der Linksvollständigkeit: Feld-Einbettung leicht möglich.

Vater und Söhne zu a[i]:



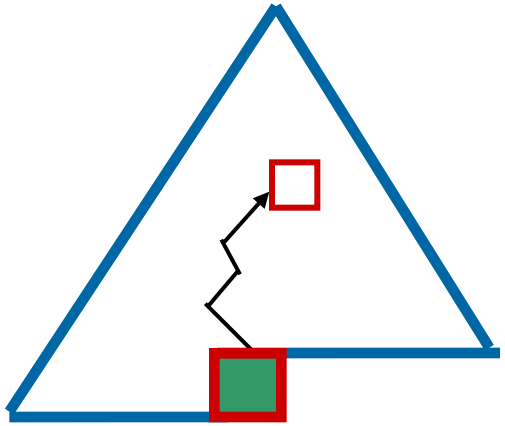
Element einfügen: upHeap(„swim“)

- neues Element in Heap einfügen
  geht (wegen Linksvollständigkeit)
  nur an genau einer Position.

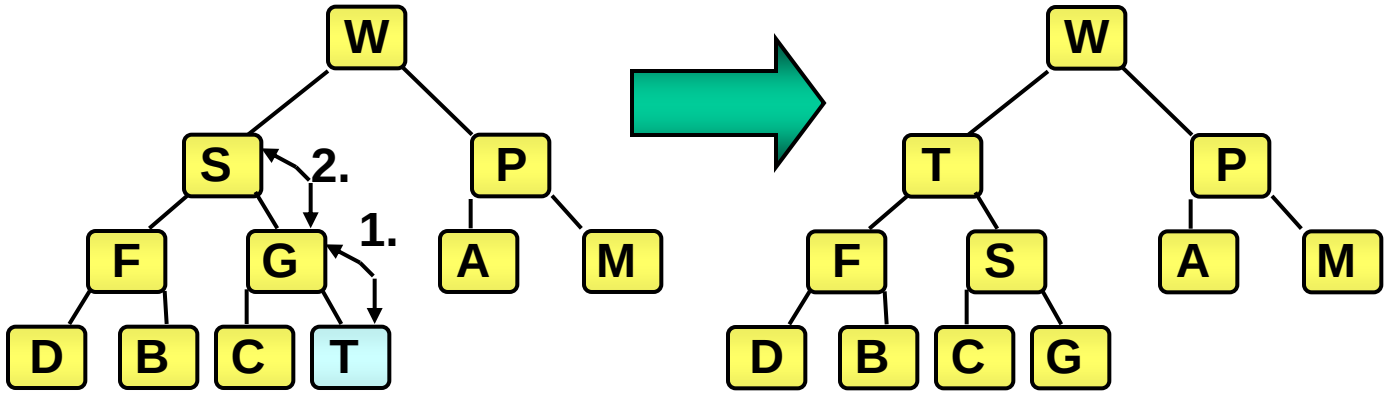
- Ordnungseigenschaft kann dadurch verletzt werden.

- Algorithmus zum Wiederherstellen der Ordnung: upHeap.

- Idee:
  - Knoten mit Vaterknoten vergleichen und ggf. vertauschen.
  - Dies setzt sich nach oben fort (notfalls bis zur Wurzel).



upHeap: Beispiel



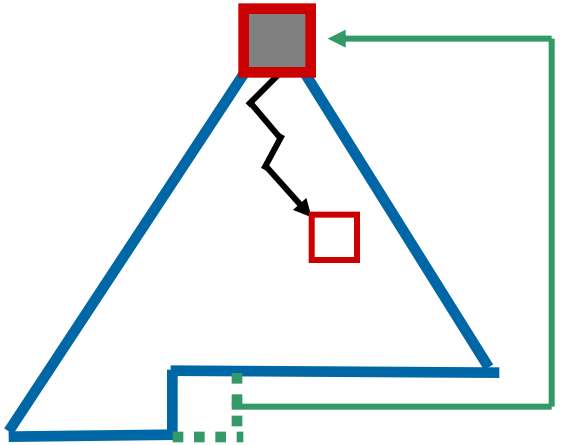
Wurzel entfernen: downHeap („sink“)

- Nach Entfernen der Wurzel eines Heaps wird am weitesten rechts stehender Blattknoten in der untersten Ebene die neue Wurzel.
  ⇒ Form wiederhergestellt.

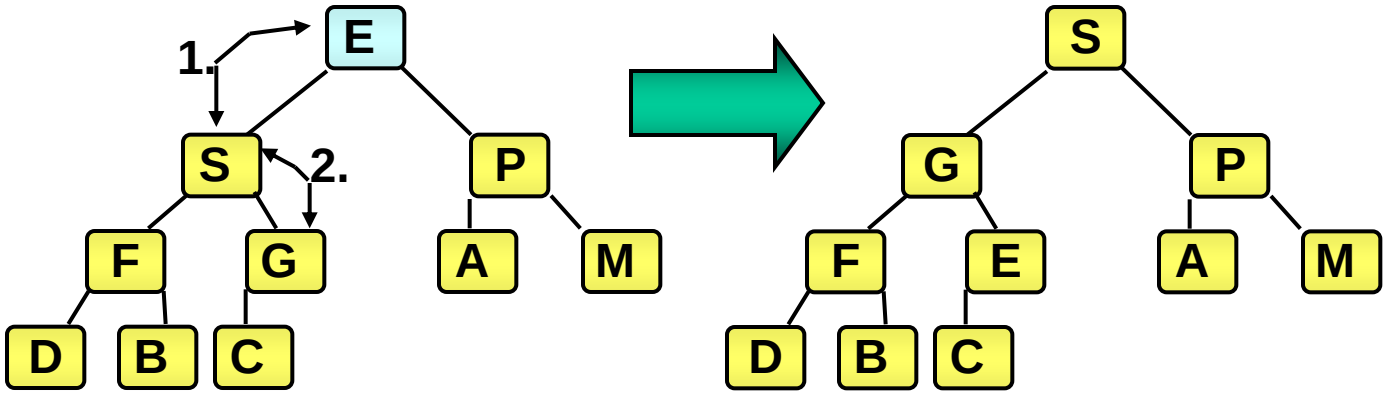
- Ordnungseigenschaft kann dadurch verletzt werden.

- Algorithmus zum Wiederherstellen der Ordnung: downHeap.

- Idee:
  - Neues Wurzel-Element wandert nach unten („versickert“).
  - Dabei wird es jeweils falls nötig mit dem größeren Sohn vertauscht.



downHeap: Beispiel (1)



Aufbau eines Heaps

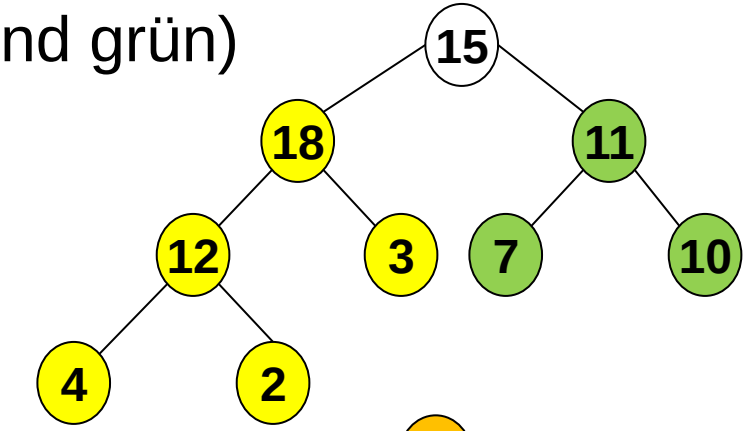
- Einfache Methode:
  - **Alle Elemente der Reihe nach einfügen und aufsteigen lassen.**

- Falls ein bestehendes (unsortiertes) Array in einen Heap umgewandelt werden soll, ist das **Bottom-up-Verfahren** effizienter:
  - **Die erste Hälfte der Elemente wird nach unten versickert (downHeap).**
  - **Das Verfahren wird kurz vorgestellt.**
  - **Bei den Übungen von Hand benutzen wir es aber NICHT.**

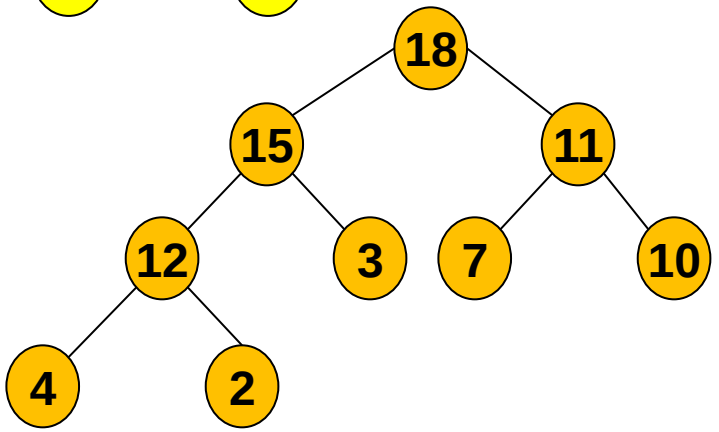
Bottom-up-Verfahren 1

Das Bottom-up-Verfahren beruht auf folgender Überlegung:

- Ausgangspunkt: 2 Heaps (gelb und grün) sind durch einen gemeinsamen Elternknoten verbunden.



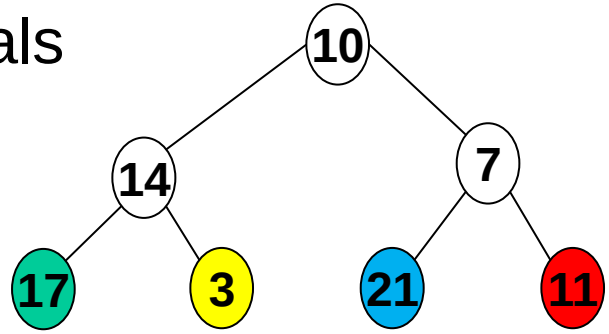
- Lässt man den Elternknoten versickern, verschmelzen beide Heaps zu einem einzigen.



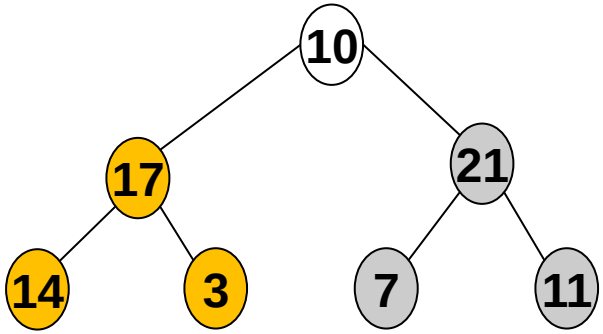
Bottom-up-Verfahren 2

Wir starten mit einem unsortierten Baum.

- Die unterste Ebene denkt man sich als viele kleine Heaps mit je einem Element.



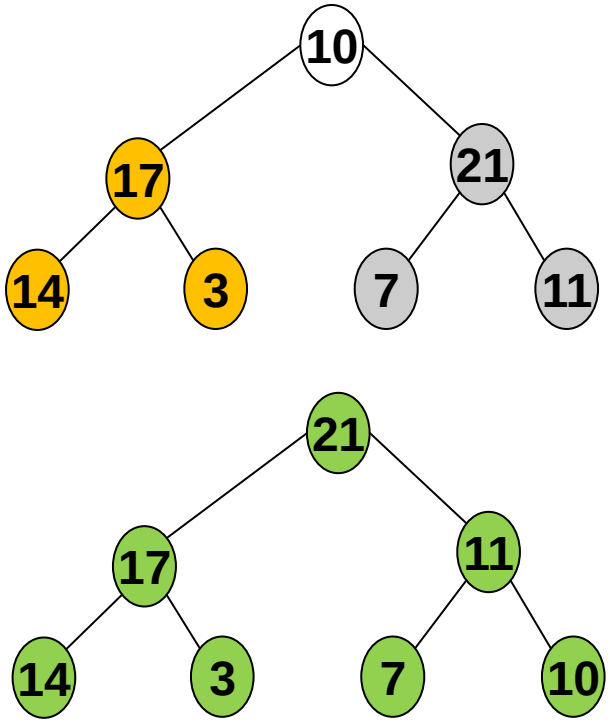
- Diese Heaps verbindet man paarweise durch Versickern der Elternelemente aus der nächsten Ebene.



Bottom-up-Verfahren 3

Das setzt man ebenenweise fort, bis man an der Wurzel angekommen ist.

Da man den Heap von unten nach oben aufbaut, heißt das Verfahren Bottom-Up-Verfahren.



Zeitkomplexität für Heap-Aufbau

- sink folgt dem Weg von der Wurzel eines links-vollständigen Binärbaums bis maximal zu seinen Blättern, d.h. es werden höchstens

$$\lceil \log_2(n+1) \rceil$$

Knoten besucht. (minimale Höhe +1) $\Rightarrow$ O(log n)

- Aufbau eines Heaps (mit Bottom-up) ruft sink für n/2 Elemente auf, also:

T^{av}heapcreate(n) = T^{w} heapcreate(n) =

$$n/2 \cdot \lceil \log_2(n+1) \rceil \in O(n \log n)$$

Heap-Sort

Der Heap ist die Grundlage für ein bekanntes Sortierverfahren

- Zu Beginn: Unsortiertes Feld

- Phase 1: Aufbau des Heaps
  - **Einfaches Verfahren: Alle Elemente werden nacheinander eingefügt**
  - **Schneller: Bottom-up-Verfahren.**
  - **Resultat ist ein Heap, der in ein Feld eingebettet ist.**

- Phase 2: Der Heap wird geleert, indem immer wieder die Wurzel (d.h. das größte Element) entfernt wird.
  - **Die Elemente werden in absteigender Reihenfolge entfernt.**
  - **Der Heap schrumpft immer weiter.**

Trick beim Heap-Sort

- Zunächst wird das Feld komplett in einen Heap umgewandelt:

- Wenn die Wurzel entfernt wird, schrumpft der Heap um 1 Element und das letzte Feldelement gehört nicht mehr zum Heap.
- Hier kann das 1. sortierte Element untergebracht werden

Trick beim Heap-Sort (2)

- Wenn weitere Male die Wurzel entfernt wird, schrumpft der Heap immer weiter. Die freiwerdenden Stellen werden mit den entnommenen Werten besetzt, die jetzt aufsteigend sortiert sind.

- Zuletzt ist der Heap auf 0 geschrumpft und das Feld ist sortiert.

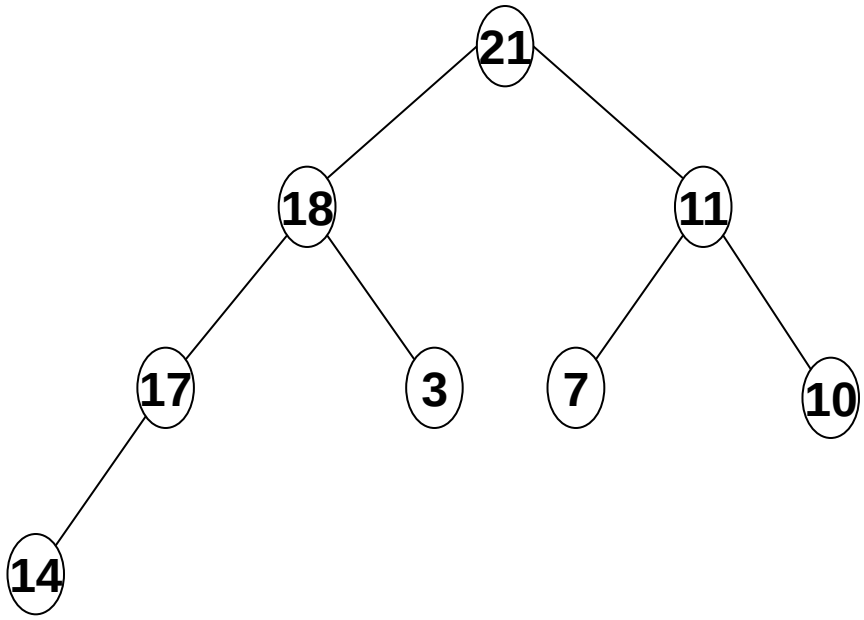
Beispiel für Phase 1 von HeapSort

18	14	7	17	3	11	10	21

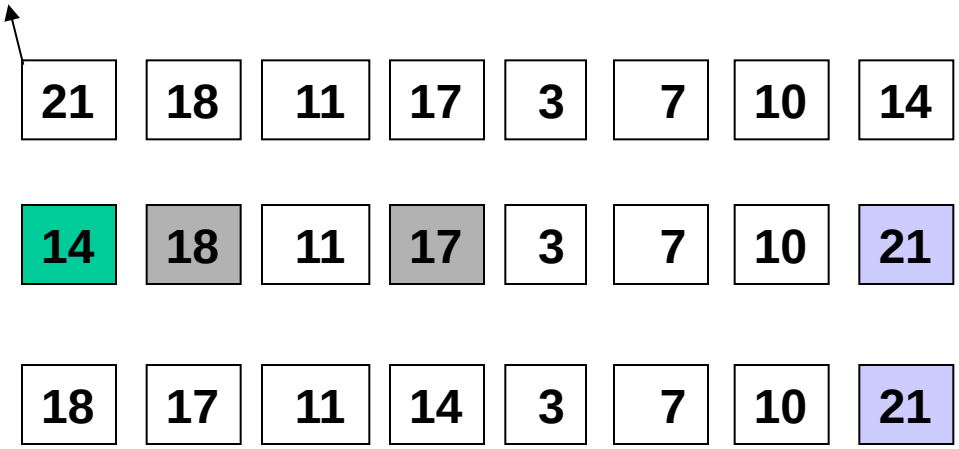
Heap

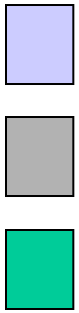
Heap aufbauen

18	14	7	17	3	11	10	21
18	14	7	17	3	11	10	21
18	14	7	17	3	11	10	21
18	17	7	14	3	11	10	21
18	17	7	14	3	11	10	21
18	17	11	14	3	7	10	21
18	17	11	14	3	7	10	21
21	18	11	17	3	7	10	14



Beispiel für Phase 2 von HeapSort

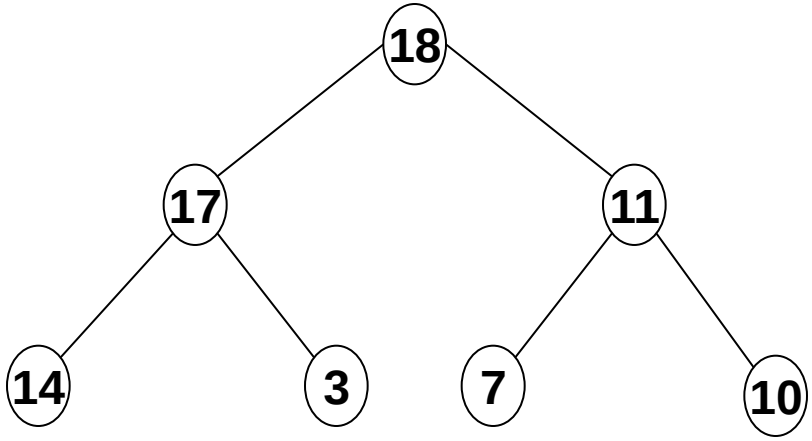




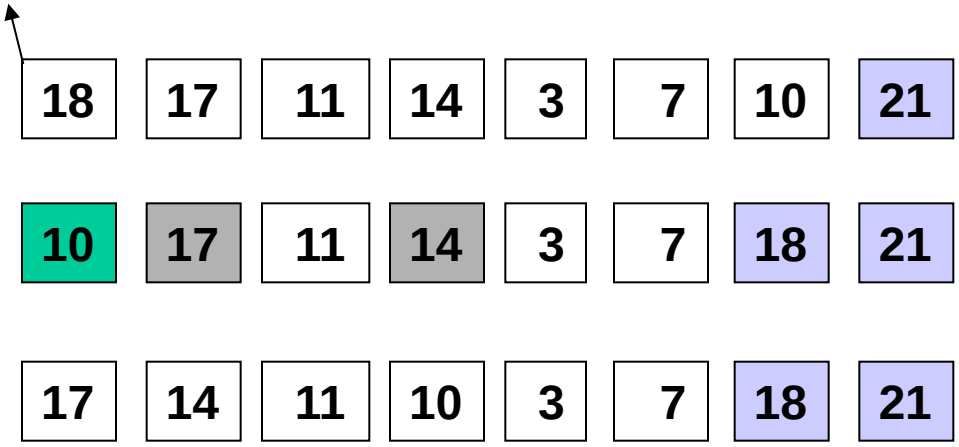
Sortierte Teilfolge

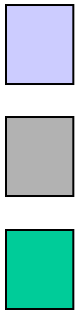
Verschobene Elemente

versickerndes Element



Beispiel für Phase 2 von HeapSort

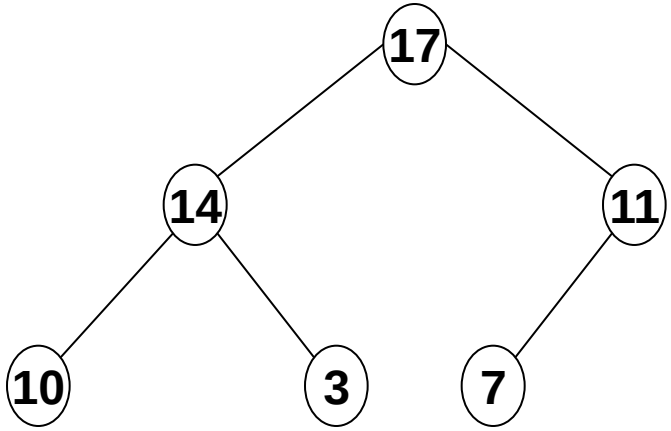




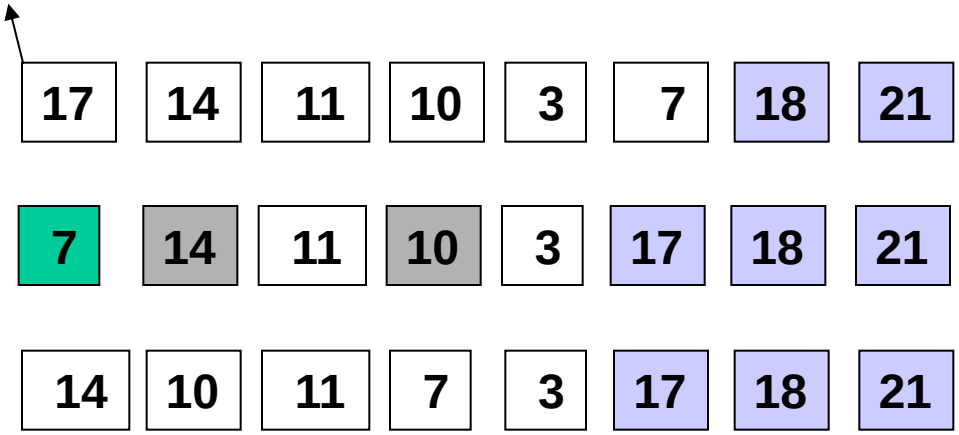
Sortierte Teilfolge

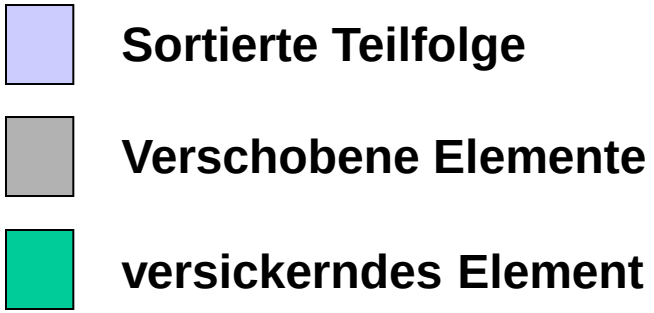
Verschobene Elemente

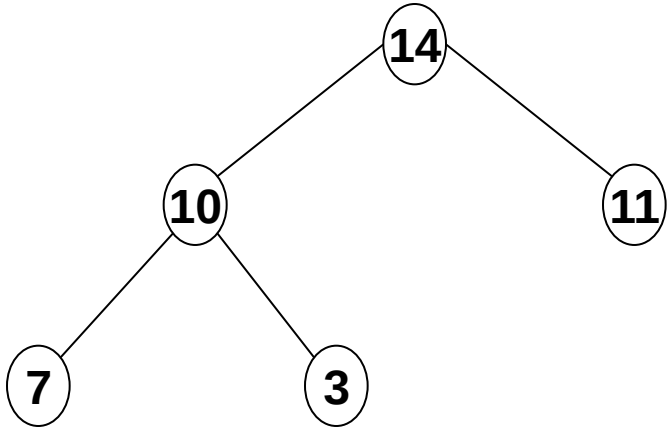
versickerndes Element



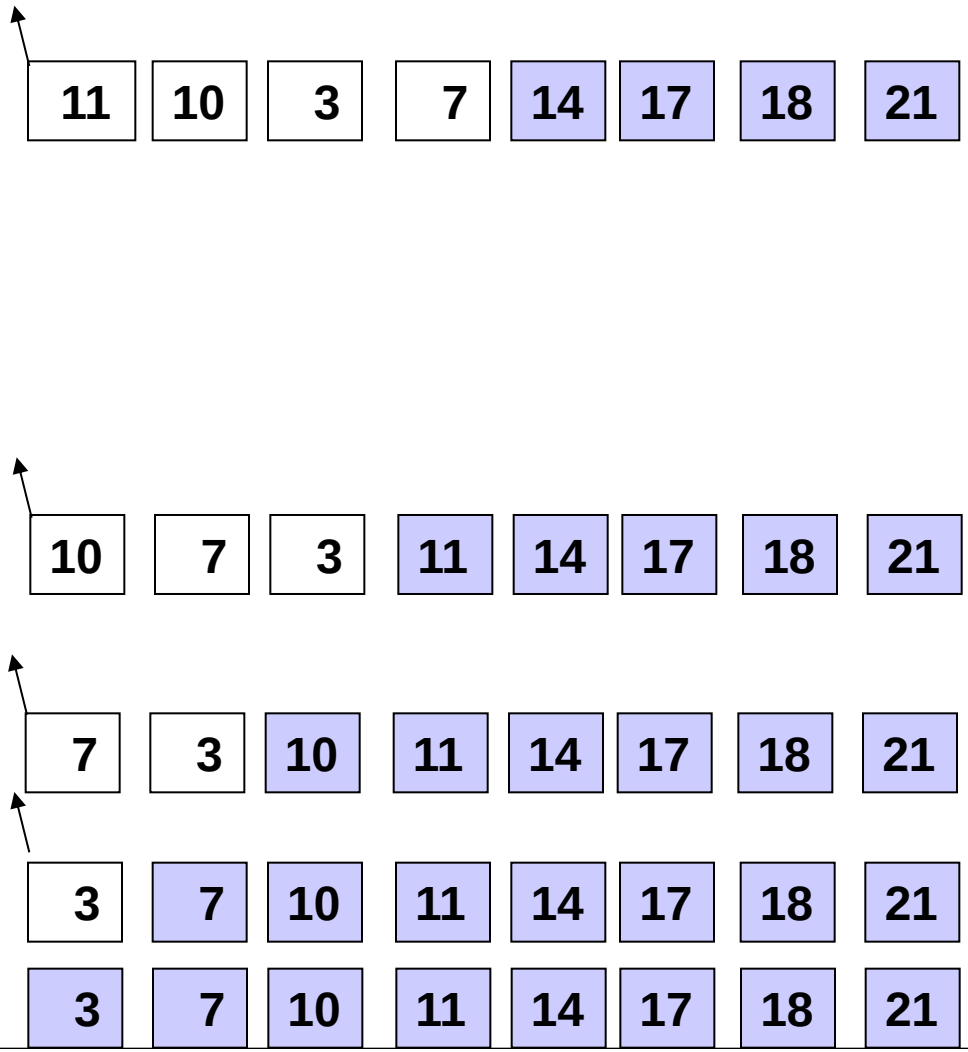
Beispiel für Phase 2 von HeapSort

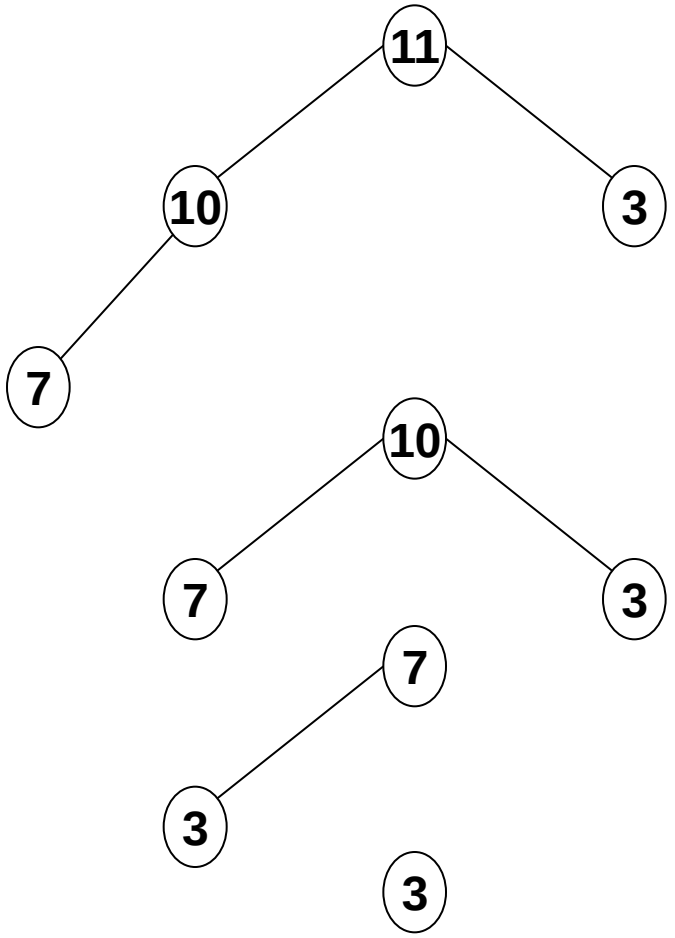






Beispiel für Phase 2 von HeapSort





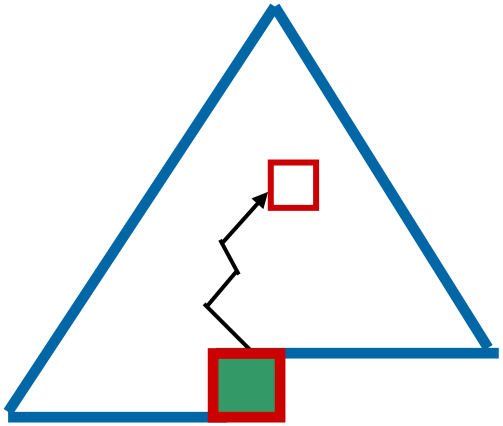
Komplexität von HeapSort

Die Laufzeitkomplexität von Heap-Sort ist **O(n log n)**

- sowohl im Best Case, als auch im Worst Case und im Average Case.
- Die Begründung ist alles andere als trivial.
- Sie wird hier nur kurz skizziert.

Komplexität von Heap-Sort im Worst Case

Upheap /Downheap (Worst Case):
Ein Upheap/Downheap setzt sich maximal über alle Ebenen des Heaps fort, ist also O(log n).



Aufbau und Abbau (Worst Case)

Wenn man beim Aufbau oder Abbau jedesmal im Worst Case ist, ergibt sich:

$T \leq \log(1)+\log(2)+\log(3)+ \ldots + \log(n) = \log(n!)$

Diese Funktion ist $\in O(n \log n)$

- Zum Beweis nähert man n! mit der Stirling-Formel an: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- Der weitere Beweis wird übersprungen.

Probleme bei der Komplexitätsberechnung

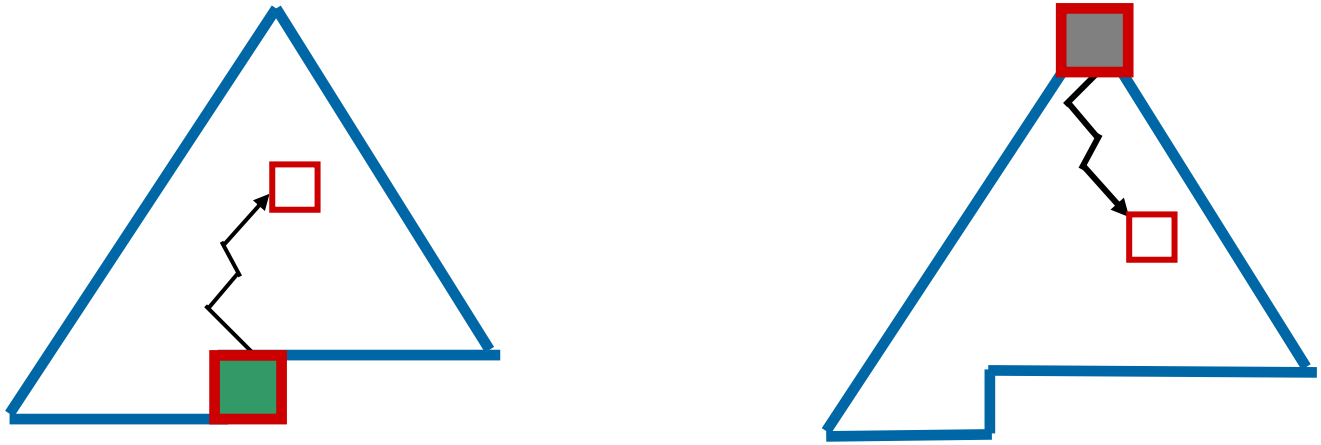
Es stellen sich folgende Fragen:

- Kann dieser Worst Case wirklich eintreten? Ist die Abschätzung vielleicht zu pessimistisch?
- Wie sieht es beim Average und beim Best Case aus? Sind sie vielleicht sogar O(n)?

Unterschiede zwischen Aufbau und Abbau

Für Heap Sort muss zunächst ein Heap aufgebaut und anschließend wieder abgebaut werden.

- Auf- und Abbau müssen getrennt betrachtet werden.
- Sie haben unterschiedliche Laufzeitverhalten.
- Man kann das gut erkennen:
  - **In den unteren Ebenen des Heaps sind mehr Elemente als in den oberen.**
  - **Ein Upheap geht durchschnittlich über weniger Ebenen als ein Downheap.**

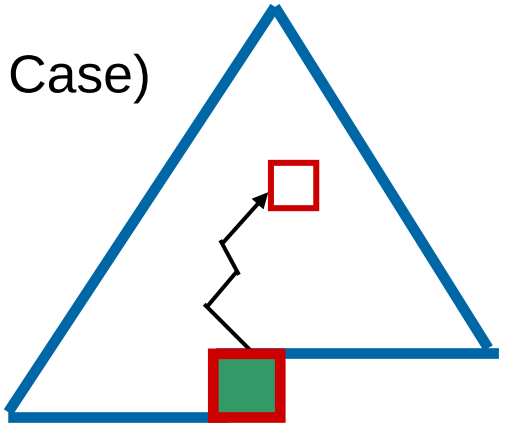


Aufbau des Heaps

Wird der Heap der Reihe nach mit Elementen befüllt,

- tritt der berechnete Worst Case ein, wenn die Elemente umgekehrt sortiert eingefügt werden.
  - **Der Worst Case ist O(n log n).**
- tritt der Best Case ein, wenn die Elemente sortiert eingefügt werden.
  - **Der Best Case ist O(n).**

- Nimmt man allerdings das Bottom-Up-Verfahren ist der Worst Case (wie auch Best- und Average Case) **O(n)**.
- Beweisidee: Man kann zeigen, dass die Summe der Länge der Pfade von allen Knoten bis in die unterste Ebene kleiner ist, als die Anzahl der Kanten im Heap.



Abbau des Heaps

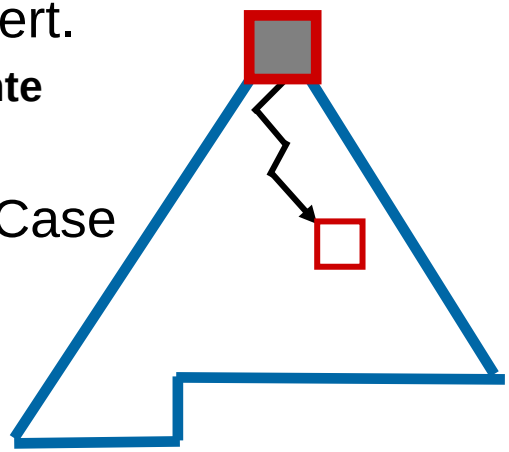
Der Abbau des Heaps ist im **Worst Case**, wenn das Feld sortiert eingefügt wurde.

- Also genau dann, wenn der Aufbau im Best Case war.

Jedes Element sickert ganz nach unten → O(n log n).

Im **Best Case** sickert nicht jedes Element ganz nach unten.

- Man kann aber zeigen, dass ein fester Prozentsatz der Elemente immer ganz nach unten sickert.
  - **Voraussetzung: Keine doppelten Elemente**
  - **Komplexer Beweis.**
- Auch der Best Case und der Average Case sind O(n log n).



Zusammenfassung

1. Aufbau des Heaps in allen Fällen:
   $T_{aufbau} \in O(n)$

2. Abbau des Heaps in allen Fällen:
   $T_{abbau} \in O(n \log n)$

$\Rightarrow$ Für HeapSort insgesamt gilt dann in allen Fällen:
$T_{heapSort}(n) \in O(n \log n)$

RWTH AACHEN UNIVERSITY

HeapSort:

- O(n log n), auch im worst case
  - **Hauptvorteil gegenüber Quick-Sort**
  - **Wird daher in Quick-Sort-Optimierung „Intro-Sort“ verwendet.**
- Kein zusätzlicher Speicher nötig.
- Nicht stabil.
- Vorsortierung wird nicht ausgenutzt.
- Im Normalfall langsamer als QuickSort.

O(n log n): Schnelle Sortiervahren

19.6 Merge-Sort

Sortieren durch Mischen: Merge-Sort

- **Prinzip:**
  - Sortiere die Daten paarweise

C: | 5 | 1 | 8_A | 7 | 3 | 8_B | 12 | 2 |

C: | 1 | 5 | 7 | 8_A | 3 | 8_B | 2 | 12 |

  - Füge jeweils zwei Paare zu sortierten Viererfolgen zusammen (Mischen)

C: | 1 | 5 | 7 | 8_A | 2 | 3 | 8_B | 12 |

  - Füge zwei Vierfolgen zu einer sortierten Achterfolge zusammen (Mischen)

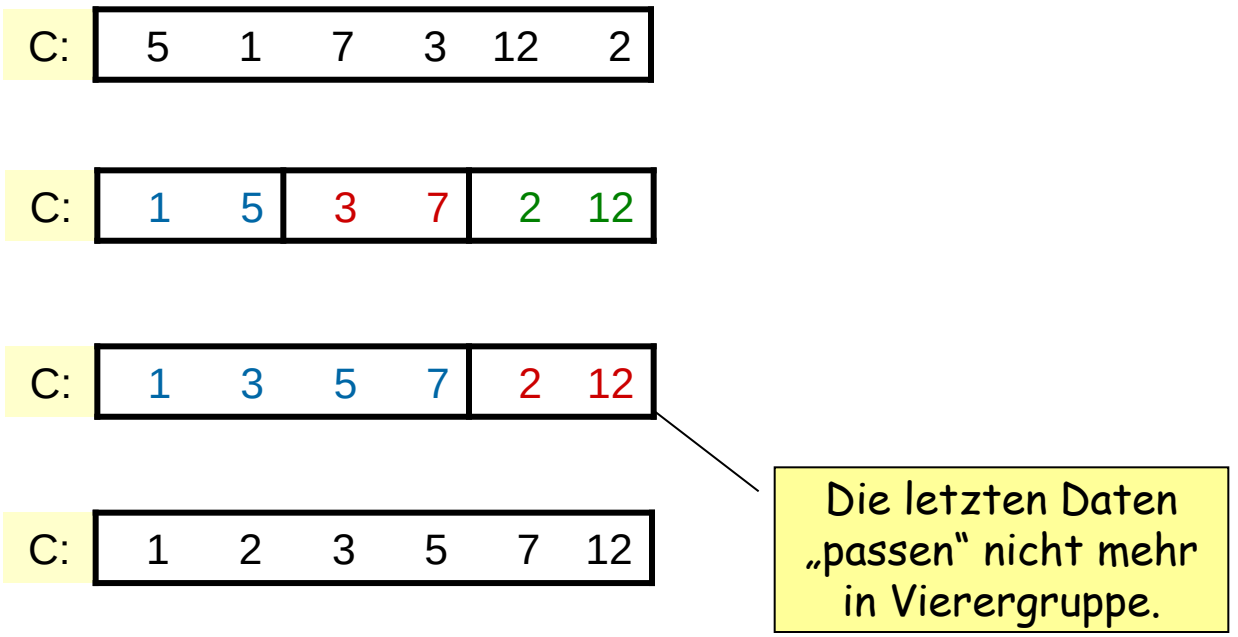
C: | 1 | 2 | 3 | 5 | 7 | **8_A** | **8_B** | 12 |

  - Usw. bei größeren Datensätzen.
  - Der eigentliche Aufwand liegt in den Mischvorgängen.

8 aus A hat Vorrang vor 8 aus B ⇒ MergeSort ist stabil.

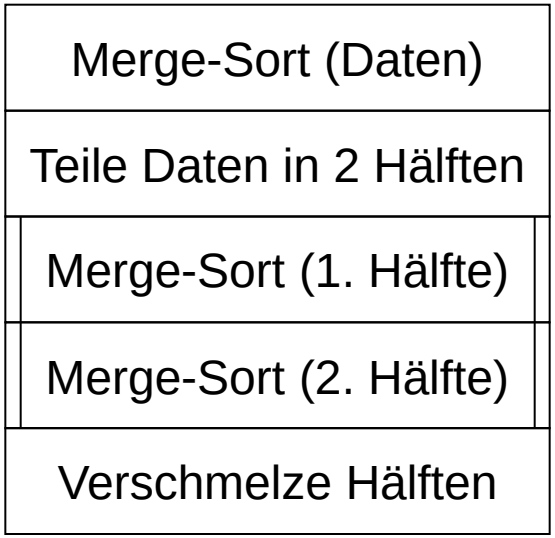
Krumme Anzahl von Datensätzen

- **Prinzip (iterative Variante):**



Iterativer / Rekursiver MergeSort

- Außer der bisher behandelten iterativen Variante gibt es auch eine rekursive Variante.
- Krumme Zahlen werden dabei gleichmäßiger aufgeteilt, wodurch die rekursive Variante etwas schneller ist.



MergeSort: Merge

Merge

Eingabe: nA sortierte Gruppen in Sequenz A,
nB sortierte Gruppen in Sequenz B,
Gruppenlänge len mit (nA+nB)*len = N

Ausgabe: N Elemente in Sequenz C
in sortierten Gruppen der Länge 2*len

Über alle Paare gA, gB von Gruppen der Länge len aus Sequenz A bzw. aus Sequenz B

Solange noch Elemente sowohl in gA als auch in gB

a = kleinstes Element aus gA, b = kleinstes Element aus gB

a <= b

Wahr	Falsch
Entnehme a aus gA und hänge es an C an	Entnehme b aus gB und hänge es an C an

Verschiebe restliche Elemente von gA nach C

Verschiebe restliche Elemente von gB nach C

Komplexitätsanalyse von MergeSort

- Sequenz C wird $k = (ld\ n)$-mal zerlegt und dann wieder zusammengemischt.

- Verteilen und Mischen erfordern jeweils $O(n)$ Operationen.

$\Rightarrow T_{mergesort}(n) \in O(n \log n)$ (auch im worst case)

Optimierung 1: Felduntergrenze

- Viele sinnvolle Optimierungsmöglichkeiten.

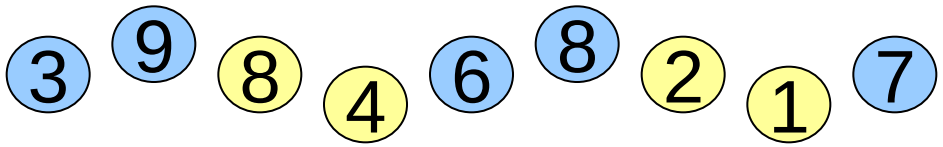
1. Feld wird nicht in Einzelelemente geteilt, sondern in Gruppen zu n Elementen, die im 1. Schritt mit InsertionSort sortiert werden.
   - **Ähnlich wie bei QuickSort.**

Optimierung 2: Trivialfall

- Java überprüft beim Zusammenfügen:
  - **ist das kleinste Element der einen Teilfolge größer ist als das größte Element der anderen Teilfolge?**
  - **wenn ja, beschränkt sich das Zusammenfügen auf das Hintereinandersetzen der beiden Teilfolgen.**
  - **Damit wird eine Vorsortierung ausgenutzt.**

Optimierung 3: Natürlicher Mergesort

- Weitergehende Ausnutzung der Vorsortierung.
- Jede Zahlenfolge besteht aus Teilstücken, die abwechselnd monoton steigend und monoton fallend sind.



Die Idee ist, diese bereits sortierten Teilstücke als Ausgangsbasis des Merge-Sorts zu nehmen.

- Bei nahezu sortierten Feldern werden die Teilstücke sehr groß und das Verfahren sehr schnell (O(n) im Best case).

Merge-Sort auf der Festplatte

- Merge-Sort auf externen Laufwerken:
  - **Teile die Daten in zwei gleich große Dateien A und B.**
  - **Lese jeweils ein Datum von A und B. Schreibe das sortierte Paar abwechselnd in zwei neue Dateien C und D.**
  - **Lese jeweils ein Paar aus C und D. Verschmelze die Paare und schreibe die Vierergruppen abwechselnd in A und B.**
  - **usw.**
  - **Wiederhole, bis in einer Datei die komplette sortierte Folge sthet.**

MergeSort:

- O(n log n), auch im Worst case
- MergeSort kann auch externe Daten sortieren.
- Stabil
- Benötigt zusätzlichen Speicherplatz.
- Durchschnittlich langsamer als Quicksort.
- Nutzt Vorsortierung nicht aus

Optimierung: Natural MergeSort

- Nutzt Vorsortierung aus.
- Bei vorsortierten Daten häufig schneller als O(n log n).

19.7 Vergleich der Sortierverfahren

Vergleich der Sortierverfahren

- Eigene Laufzeitmessungen

Verfahren	**Laufzeit 1000 Elem.**	**Laufzeit 10.000.000 Elem.**
Simple Sort	*1,2 ms*	*1d 15 h*
Insertion Sort	*0,2 ms*	*4h 20 min*
Quick Sort	*0,07 ms*	*1,2 s*

Vergleich der Sortierverfahren

Verfahren	Laufzeitmessungen (nach Wirth, Sedgewick)	Vorsortierung ausnutzen	Worst Case ok	Zus. Speicher	Stabil
QuickSort	*100*			*Φ(log n)*	
HeapSort	*150-200*		*X*	*Φ(1)*	
MergeSort	*150-200*	*X*	*X*	*Φ(n)*	*X*
MSB-Radix-Sort	*85 (bei 100.000 El.)*		*X*	*Φ(n)*	*X*

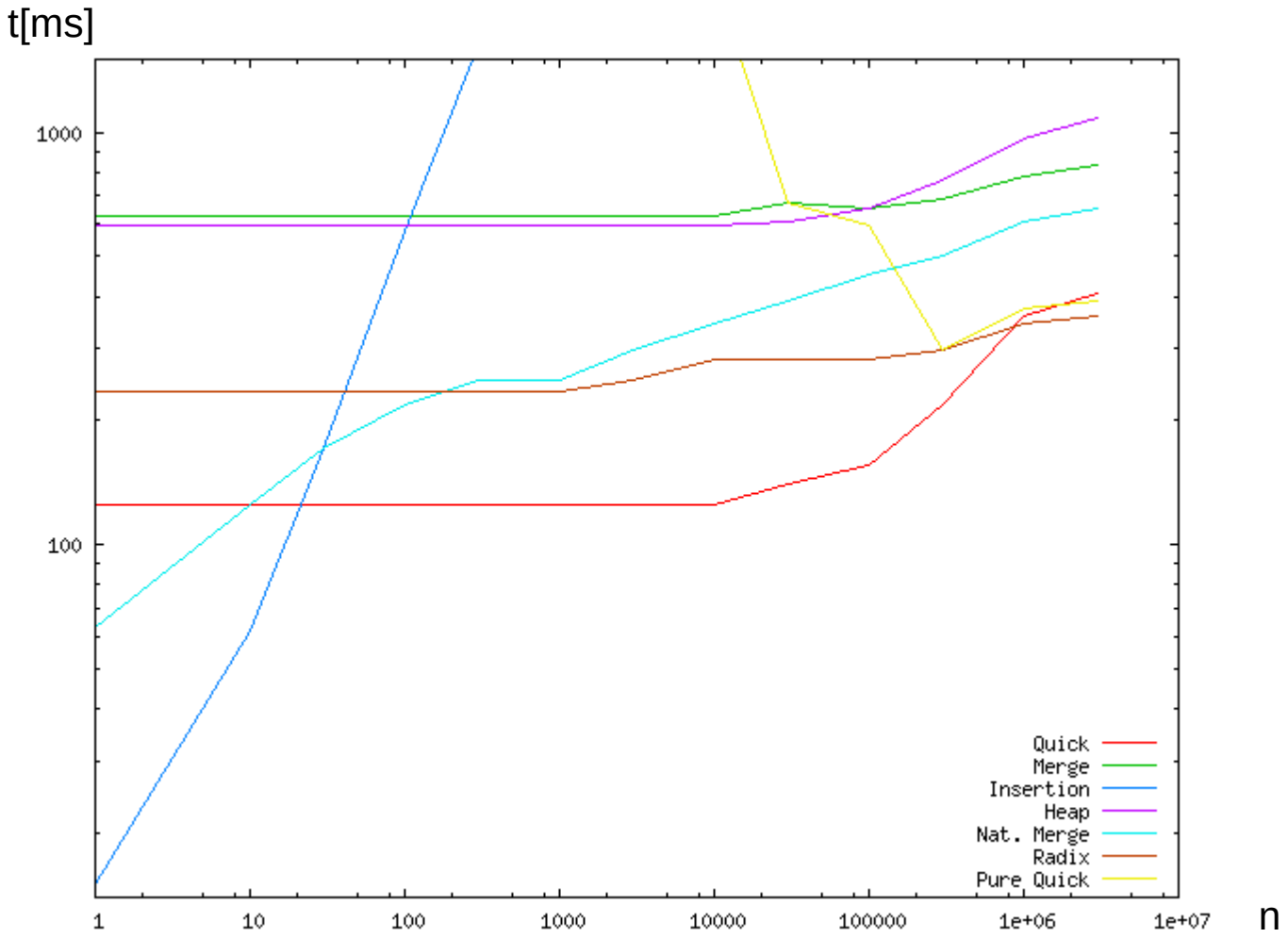
Vorsortierte Felder

- Welches Verfahren ist für nahezu sortierte Felder am besten?

- Experiment
  - **2.000.000 Elemente**
  - **Feldinhalt = Feldindex**
  - **n Elemente werden paarweise vertauscht**
  - **n variiert zwischen 0 (sortiert) und 3.000.000 (unsortiert).**

Sortierver-fahren	Sortiertes Feld	Unsortier-tes Feld
Bubble	O(n)	$O(n^2)$
Insertion	O(n)	$O(n^2)$
Quick („Pur“)	$O(n^2)$	O(n log n)
Heap, Merge, Quick	O(n log n)	O(n log n)
Nat. Merge	O(n)	O(n log n)
Radix	O(n)	O(n)

Vorsortierte Felder (2)



Abschlussfragen 1/1

1. Beschreiben Sie kurz den Algorithmus „Simple-Sort“.
2. Welche O-Klasse haben einfache und schnelle Sortierverfahren bezüglich der Laufzeit?
3. Welche sind die zwei bekanntesten einfachen Sortierverfahren?
4. Welche sind die drei bekanntesten schnellen Sortierverfahren?
5. Wann kann ein Verfahren nicht in eine allgemeine Bibliothek aufgenommen werden?
6. Was bedeutet „Stabilität“?
7. Warum kann ein Verfahren bei wenigen Elementen schneller sein und ein anderes bei vielen Elementen?
8. Welches ist das meistgenutzte Sortierverfahren?
9. Welches ist das beste einfache Sortierverfahren?
10. Wie ist die Laufzeitkomplexität dieses Verfahrens im „Best Case“?

Abschlussfragen 1/2

11. Welches Entwurfsprinzip implementiert Quick-Sort?
12. Was ist ein Pivot-Element?
13. Welche Zeitkomplexität hat Quick-Sort im Worst Case?
14. Bei welcher Wahl des Pivots tritt der Worst Case gerade bei schon sortierten Listen auf.
15. Welche Optimierung bei der Wahl des Pivots ist üblich?
16. Welche zweite Optimierung wird gewöhnlich bei Quick-Sort verwendet?
17. Welche Quick-Sort-Variante wird in Java verwendet?
18. Wann und warum wird in Java Quick-Sort nicht verwendet?

Abschlussfragen 2/1

1. Welche beiden Hauptfunktionen umfasst die ADT Prioritätswarteschlange?
2. Mit welcher Datenstruktur aus den vergangenen Vorlesungen kann man eine Prioritätswarteschlange implementieren?
3. Welche beiden Eigenschaften muss ein Binärbaum erfüllen, um ein Heap zu sein?
4. An welcher Stelle werden neue Elemente in einen Heap eingefügt? Was passiert anschließend mit den Elementen?
5. Beim Entnehmen des obersten Elements bleibt eine Lücke zurück. Wie stopft man die Lücke? Was passiert anschließend?
6. Wie ist die Laufzeitkomplexität beim Heap für Einfügen und Entnehmen im Average Case?
7. Worin besteht der Vorteil eines Heaps gegenüber anderen Datenstrukturen für die ADT Prioritätswarteschlange?

Abschlussfragen 2/2

8. Wie kommt man bei der Feldeinbettung eines Heaps an den Eltern- bzw. die Kindknoten heran?
9. Wo werden beim Heap-Sort die entnommenen Elemente abgelegt?
10. Welche Laufzeitkomplexität hat Heap-Sort im Best-, Worst- und Average-Case?
11. Welchen anderen großen Vorteil hat das Heap-Sort-Verfahren?
12. Welches Entwurfsprinzip nutzt Merge-Sort?
13. An welcher Stelle unterscheidet sich der iterative vom rekursiven Merge-Sort?
14. Welche der beiden Varianten ist schneller?
15. Beschreiben Sie kurz den Algorithmus des rekursiven Merge-Sorts.
16. Welche Optimierung ist bei Merge-Sort üblich?
17. Welche besonderen Vorteile hat der natürliche Merge-Sort?