Suchen und Sortieren
(Die klassischen Algorithmen)

- **Lineare Suche** und **Binäre Suche** (Vorbedingung und Komplexität)
- Sortieralgorithmen (allgemein)
- **Direkte Sortierverfahren** (einfach aber langsam)
- Schnelle Sortierverfahren (**Mergesort, Quicksort**)
- Schnelles Sortieren in Bäumen (**Heapsort**)

... noch schnelleres spezielles Verfahren Proxmap

Wie soll die Schnittstelle einer Suchfunktion aussehen?

Problem: finde Index eines vorgegebenen Objekts in einem Array

a) **Wertdaten**

```
int[] feld1 = {17, .. };
int x = FindLinear.find(feld1, 14);
```

(Vergleiche mit ==)

b) **Referenzdaten***

```
Integer[] feld2 = {Integer.valueOf(17), ...};
int x = FindLinear.find(feld2, Integer.valueOf(14));
```

(Vergleiche mit `equals()`-Methode)

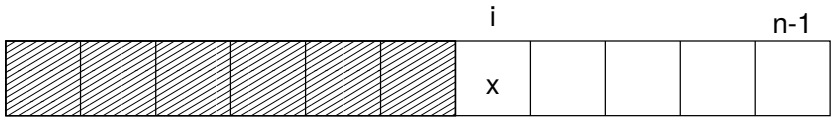
Anmerkung: schnelle Suchverfahren erfordern auch < -Vergleiche.

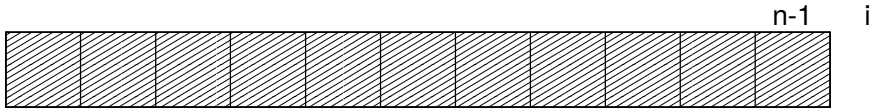
*wg. Autoboxing kann man das auch so schreiben (identischer Effekt):

```
Integer[] feld2 = {17, …}; //Array mit Referenzdaten (Integer)
int x = FindLinear.find(feld2, 14);
```

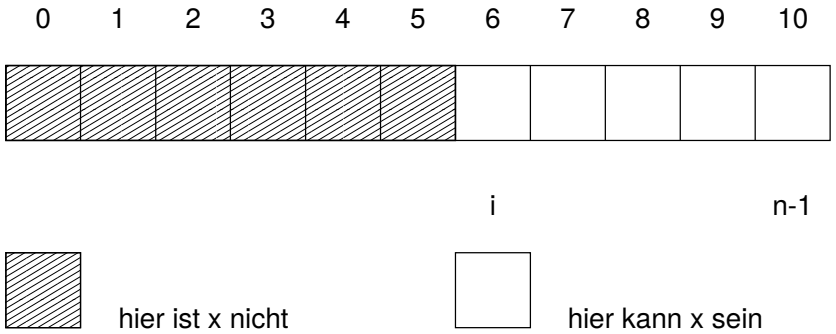
Die Idee der linearen Suche

Ziel:





Invariante:



Lineare Suche ist ganz einfach.

```
/**
 * Finde die Position einer ganzen Zahl in einem Feld.
 * @param a Feld in dem gesucht wird
 * @param key zu suchende Zahl
 * @return wenn gefunden die Position, sonst -1;
 */
public static int find(int[] a, int key) {
    // Invariante: key ist nicht in a[0:i-1]
    // Abbruchbedingung: i=a.length oder a[i]=key
    int i = 0;
    while (i != a.length && key != a[i]) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i = a.length oder a[i] = key)
    return i != a.length ? i : -1;
}
```

Laufzeitverhalten der linearen Suche ist O(*n*)

Lineare Suche bei einem Feld von Referenzen*

```
/**
 * Finde die Position eines Objekts in einem Feld.
 * @param a Feld in dem gesucht wird
 * @param key zu suchendes Objekt
 * @return wenn gefunden die Position, sonst -1;
 */
public static int find(Object[] a, Object key) {
    // Invariante: key ist nicht in a[0:i-1]
    // Abbruchbedingung: i = a.length oder a[i] = key
    int i = 0;
    while (i != a.length && !key.equals(a[i])) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i = a.length oder a[i] = key)
    return i != a.length ? i : -1;
}
```

Umgang mit Null (*eine* Möglichkeit!)

```
private static final NULL_OBJECT = new Object() {
    public boolean equals(Object other) {
        return other == this || other == null;
}

public static int find(Object[] a, Object key) {
    if (key == null) key = NULL_OBJECT;
    int i = 0;
    while (i != a.length && !key.equals(a[i])) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i = a.length oder a[i] = key)
    return i != a.length ? i : -1;
}
```

Bei Comparable-Objekten geht auch compareTo()

```
/**
 * Finde die Position eines Objekts in einem Feld.
 * @param a Feld in dem gesucht wird
 * @param key zu suchendes Objekt
 * @return wenn gefunden die Position, sonst -1;
 */
public static int find(Comparable[] a, Comparable key) {
    // Invariante: key ist nicht in a[0:i-1]
    // Abbruchbedingung: i = a.length oder a[i] = key
    int i = 0;
    while (i != a.length && key.compareTo(a[i])!= 0) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i=na.length oder a[i]=key)
    return i != a.length ? i : -1;
}
```

Hier bringt `compareTo()` nur Nachteile! Bei besseren Suchverfahren brauchen wir aber aber Größenvergleiche (Sortieren sowieso).

Die Schnittstelle Comparator kann eine flexible Alternative sein!

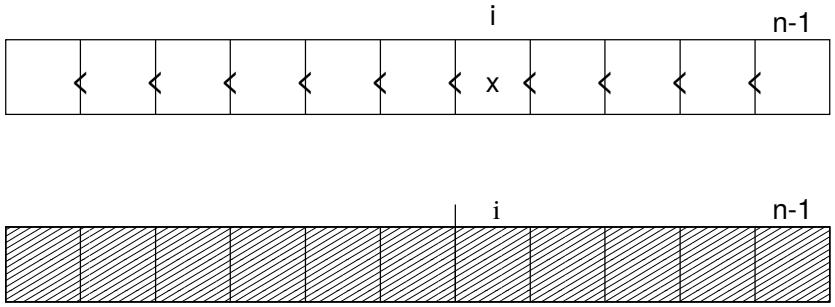
```
/**
 * Finde die Position eines Objekts in einem Feld.
 * @param a Feld in dem gesucht wird
 * @param key zu suchendes Objekt
 * @param comp Comparator-Objekt, das die Ordnung festlegt
 * @return wenn gefunden die Position, sonst -1;
 */
public static int find(Object[] a, Object key, Comparator comp) {
    // Invariante: key ist nicht in a[0:i-1]
    // Abbruchbedingung: i = a.length oder a[i] = key
    int i = 0;
    while (i != a.length && comp.compare(key, a[i])!=0) {
        // key nicht in a[0:i]
        i++;
    }
    // Ziel: key ist nicht in a[0:i-1]
    //       und (i = a.length oder a[i] = key)
    return i != a.length ? i : -1;
}
```

Durch ein Comparator-Objekt kann von Fall zu Fall festgelegt werden, was Gleichheit oder was Größenvergleich bedeutet.

Die binäre Suche ist schneller, aber setzt sortiertes Feld voraus

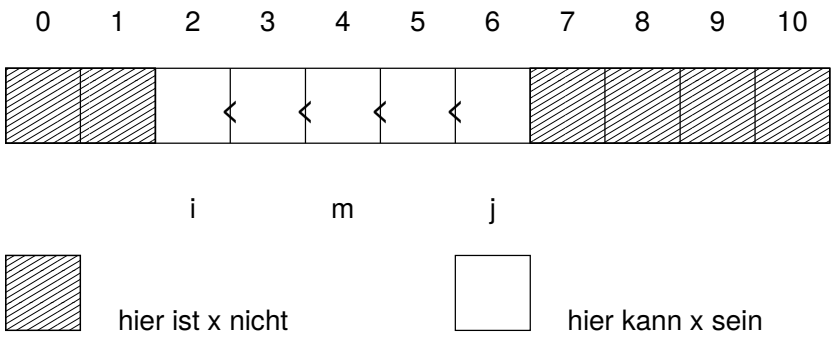
Vorbedingung: das Feld ist sortiert

Ziel:



Rückgabe: -i-1

Invariante:



Binäre Suche verlangt sortiertes Feld ... und ist relativ schnell.

```
/** finde Position von Object x in Feld a
 * @param a sortiertes Feld von Objekten
 * @param key zu suchendes Objekt (bzw. key des Objekts)
 * @param cmp Comparator für die Suche des Objekts
 * @return wenn gefunden: Index des Objekts, sonst -(index+1) vor
 *         dem key stehen sollte.
 */
public static int find(Object[] a, Object key, Comparator cmp) {
    /* Vorbedingung: n >= 0 und a ist aufsteigend sortiert
     * Ziel: (key ist nicht in a[0:n-1] und i>j) oder a[m]=x
     * Invariante: key ist höchstens in a[i:j]
     */
    int i = 0;           // untere Intervallgrenze
    int j = a.length-1; // obere  Intervallgrenze
    while (i <= j) {
        int m = (i+j)/2;     // Intervallmitte
        int compareResult = cmp.compare(key, a[m]);
        if (compareResult == 0) return m:
        else if(ccompareResult <  0) j = m-1;
        else i = m+1;
    }
    return -i-1;
}
```

Laufzeitverhalten der binären Suche ist O(log *n*)

by the way ... Rekursion ist noch einfacher

```
/** finde Position von Object x in Feld a
 * @param a sortiertes Feld von Objekten
 * @param key zu suchendes Objekt (bzw. key des Objekts)
 * @param cmp Comparator für die Suche des Objekts
 * @param i untere Grenze des Suchintervalls
 * @param j obere Grenze des Suchintervalls
 * @return wenn gefunden: index des Objekts, sonst -(index+1) vor
 *         dem key stehen sollte.
 */
public static int find(Object[] a, Object key, Comparator cmp
                       int i, int j) {
    if (i > j) return -i-1;
    int m = (i + j) / 2;     // Intervallmitte
    int compareResult = cmp.compare(key, a[m]);
    if (compareResult == 0)
        return m;
    else if (compareResult < 0)
        return find(a, key, cmp, i, m - 1);
    else
        return find(a, key, cmp, m + 1, j);
}
```

Laufzeitverhalten der binären Suche ist O(log *n*)

Sortieralgorithmen

Definition: Eine sortierte Reihenfolge ist eine Permutation der Elemente einer sequentiellen Datenstruktur, die einer bestimmten Ordnungsbeziehung genügt.

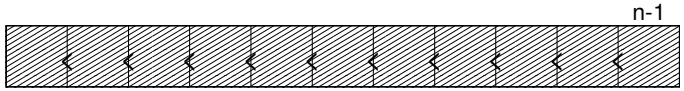
Sortieren von Feldern. (internes Sortieren)
Sortieren von Listen. (wird manchmal auf Felder zurückgeführt => intern)
Sortieren von Dateien. (externes Sortieren)

Die Tatsache, dass ein Sortierverfahren eine Permutation darstellt, erlaubt es, grundsätzlich ohne Hilfsfelder auszukommen. Meist wird internes Sortieren auf Vertauschen zweier Elemente zurückgeführt.

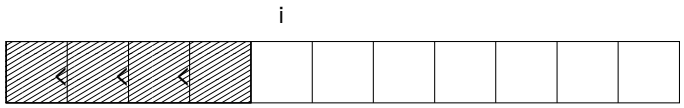
Komplexität: mindestens (!) O(n)

Direkte Verfahren sind leicht zu verstehen
-- aber zu langsam für richtige Anwendungen

Ziel:



Invarianten:

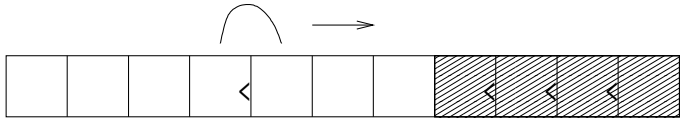




fertig sortiert: direkte Auswahl



geordnet: direktes Einfügen





fertig sortiert: direktes Vertauschen (Bubblesort)

Alle diese Verfahren sind $O(n^2)$

selection sort

insertion sort

bubble sort

Die Invariante bestimmt den Algorithmus !

Direkte Auswahl

5	7	**1**	3	8	4	2	6
1	|	7	5	3	8	4	**2**	6
1	2	|	5	**3**	8	4	7	6
			.					
			.					
			.					

Direktes Einfügen

5	|	7	1	3	8	4	2	6
5	7	|	**1**	3	8	4	2	6
1	5	7	|	3	8	4	2	6
			.					
			.					
			.					

Direkte Auswahl

```
import java.util.Comparator;

public class Selectionsort {
    public static void sort(Object[] a, Comparator comp) {
        int n = a.length;
        for (int i = 0 ; i < n - 1; i++) {
            // fertig-geordnet(a[0:i-1])
            int j = i;
            for (int k=i+1; k < n; k++ ) {
                if (comp.compare(a[k], a[j]) < 0 ) j = k;
            }
            // i<=j<n und a[j] = Mininum(a[i:n-1])
            if (i != j) {
                Object t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
            // fertig-geordnet(a[0:i])
        }
        // Q: fertig-geordnet(a[0:n-1])
    }
}
```

Vergleiche: O(n^2) + Vertauschungen: O(n) = O(n^2)

Direkte Auswahl

```
import java.util.Comparator;

public class Selectionsort {
    public static void sort(Object[] a, Comparator comp) {
        for (int i = 0 ; i < a.length - 1; i++) {
            // fertig-geordnet(a[0:i-1])
            swap(a, i, minPos(a, i, comp));
            // fertig-geordnet(a[0:i])
        }
        // Q: fertig-geordnet(a[0:a.length-1])
    }

    private int minPos(Object[] a, int from, Comparator c) {
        int m = from;
        for (int i = from + 1; i < a.length; i++) {
            if (a.compare(a[i], a[m]) < 0) m = i;
        }
        return m;
    }
    ...
}
```

Idee: rekursive Zerlegung und getrenntes Sortieren

Man kann ein Feld mit *n* Elementen ld(**n**) mal halbieren.
(1 x 8 $\rightarrow$ 2 * 4 $\rightarrow$ 4 *2 $\rightarrow$ 8*1)

Am Ende braucht man nichts mehr zu tun.

Wenn man bei jeder Zerlegung (vorher oder nachher) einen Aufwand von O(n) treibt (damit nachher alles sortiert ist) ist der Algorithmus O(n ld(n)).

Es gibt 2 Varianten:

- **Quicksort**: erst Datenelemente richtig auf 2 Hälften verteilen und dann getrennt sortieren. (alle Daten des linken Teils sind kleiner als die des rechten Teils)
- **Mergesort:** erst die beiden Teile getrennt sortieren und dann die Ergebnisse mischen.

Quicksort ist ein rekursiver Algorithmus

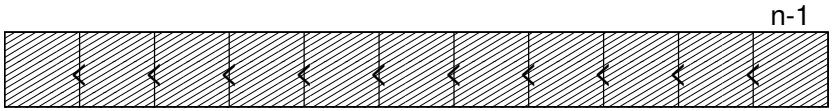
```
public static void sort(Object[] a, Comparator comp) {
    sort(a, 0, a.length-1, comp);
}

public static void sort(Object a[], int i0, int i1, Comparator comp) {
    // i0, i1 = Intervallanfang, -ende
    if (i0 < i1) {
        int mitte = partition(a, i0, i1, comp);  // O(n)
    // a[i0:i-1] <= a[mitte] < a[mitte+1:i1]
        sort(a, i0, mitte-1, comp);
        // fertig_sortiert(a[i0:mitte])
        sort(a, mitte+1, i1, comp);
        // fertig_sortiert(a[i0:i1])
    }
}
```

Wenn partition() O(n) ist <u>und</u> in der Mitte teilt, ist das Sortieren O(*n* log *n*) !

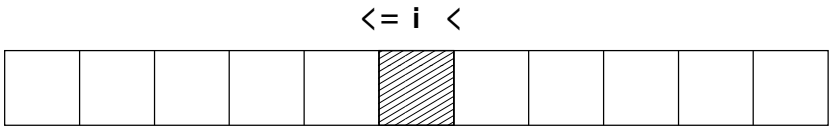
Neue Idee: Zerlegung eines Array und getrenntes Sortieren

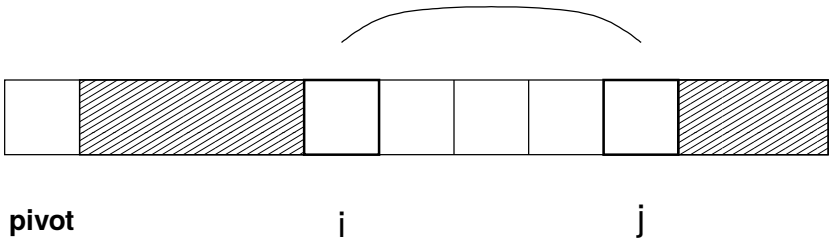
Ziel:



Ideal jeweils Halbierung
und O(n) pro Ebene:
log *n* Ebenen, also *n* log *n*

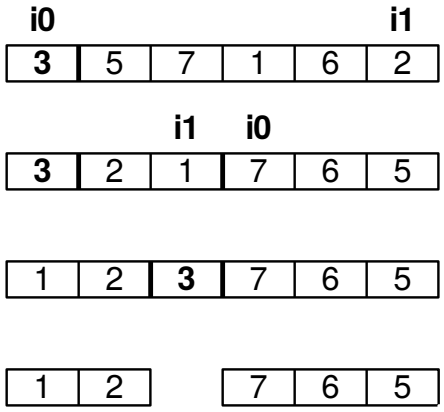
Zerlegungsidee





Algorithmus zum Zerlegen:

1. Wähle geeignetes Pivotelement
2. Tausche Pivotelement an den Anfang
3. Wandere mit i0 und i1 nach Innen und tausche bei Bedarf
4. Tausche Pivotelement in die Mitte
5. Danach ist das Pivotelement an der richtigen Stelle und links und rechts kann unbhängig sortiert werden.



Der Kernalgorithmus ist die Zerlegung

```
private static int partition(Object[] a, int i0, int i1, Comparator comp) {
    swap(a, i0, i0 + (int)(Math.random()*(i1-i0)));
    Object pivot = a[i0++];
    // Inv. (I0, I1 = Anfangswerte): a[I0:i0-1] <= pivot < a[i1+1:I1]
    while (i0 <= i1) {
        if (comp.compare(a[i0], pivot) <= 0)          // links ok
            i0++;
        else if (comp.compare(pivot, a[i1]) < 0)      // rechts ok
            i1--;
        else {                                        // tausche links-rechts
            // a[i0] > pivot >= a[i1]
            swap(a, i0, i1);
            // a[i0] <= pivot < a[i1]
            i0++; i1--;
        }
    }
    // Jetzt sind i0 und i1 um 1 zu weit gelaufen.
    //   a[I0+1:m] <= pivot < a[m+1:I1]
    // bringe pivot an die richtige Stelle
    swap(a, pivot, i1);
    return i1;
}
```

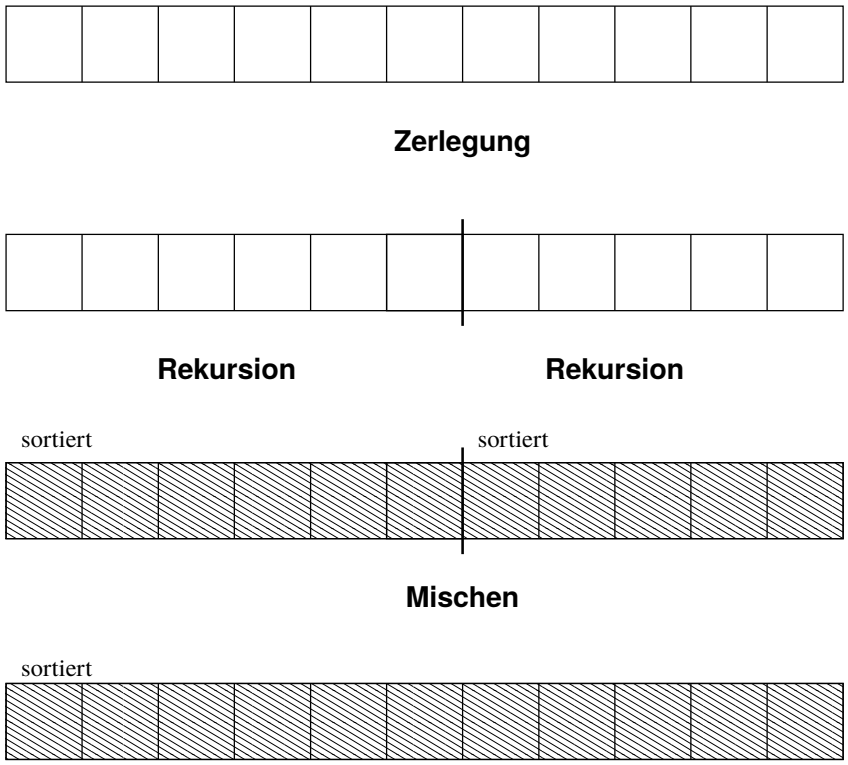
Die Zerlegung ist garantiert O(*n*), aber Halbierung ist nicht garantiert !

Übliche Optimierung: **`while`** anstelle von **`if`** und **`else if`**

Bewertung des Quicksort

- Quicksort ist mit das älteste bekannte schnelle Sortierverfahren (Anthony Hoare)
- Im Durchschnitt erreicht Quicksort O(n log n)
- Im schlechtesten Fall ist Quicksort O(n^2)
- Wenn man zur Zerlegung stets das erste Feldelement wählt, tritt der schlechteste Fall bei einem bereits sortierten Feld auf.
- Es gibt Algorithmen, die den schlechtesten Fall unwahrscheinlicher machen (Pseudomedian)
- Wenn man ein zufälliges Feldelement wählt, ist bei großen Feldern der schlechteste Fall *statistisch* ausgeschlossen (randomisierte Algorithmen)

Mergesort ist das Gegenstück zu Quicksort !



Aufwand:
log *n* Ebenen, also *n* log *n*

Einziges Problem: Mischen erfordert ein Hilfsfeld!

Mergesort ist das Gegenstück zu Quicksort

```
public static void sort(Object[] a, Comparator comp) {
    sort(a, 0, a.length-1, comp);
}

//Q: ist_sortiert(a[i0:i1])
public static void sort(Object a[], int i0, int i1, Comparator comp) {
    // i0, i1 = Intervallanfang, -ende
    if (i0 < i1) {
        int mitte = (i0 + i1) / 2;
        sort(a, i0, mitte, comp);
        sort(a, mitte+1, i1, comp);
        // ist_sortiert(a[i0:mitte]) und ist_sortiert(a[mitte+1]:i1])
        Object[] hilf = Kopie von a;
        merge(hilf, a, io, i1, mitte, comp);  // O(n)
    }
}
```

Verbesserung: unmittelbar zu Beginn ein Hilfsfeld als Kopie von a anlegen!
Da das Hilfsfeld stets eine Kopie von a ist, kann man weiteres Kopieren vermeiden, indem bei jedem Aufruf `a` und `hilf` ihre Rolle tauschen.
(siehe Praktikum).

Mergesort mit einem Hilfsfeld

(leicht verständlich mit Vor- und Nachbedingung)

```
public static void sort(Object[] a, Comparator comp) {
    Object[] b = Arrays.copyOf(a, 0, a.length);
    sort(b, a, 0, a.length-1, comp);
}

//P: s[i0:i1] == d[i0:i1]
//Q: ist_sortiert(d[i0:i1])
public static void sort(Object[] s, Object[] d,
    int i0, int i1, Comparator comp) {
    // i0, i1 = Intervallanfang, -ende
    if (i0 < i1) {
        int mitte = (i0 + i1) / 2;
        sort(d, s, i0, mitte, comp);
        sort(d, s, mitte+1, i1, comp);
        // ist_sortiert(s[i0:mitte]) und ist_sortiert(s[mitte+1]:i1])
        merge(s, d, io, i1, mitte, comp); ]
    }
}
```

Das eigentliche Mischen

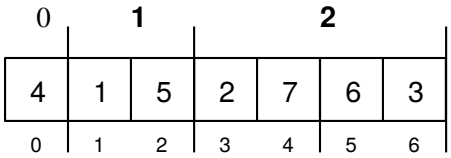
Das ist ein einfacher Kopieralgorithmus, der im direkten Vergleich entscheidet, aus welcher Feldhälfte das nächste Element genommen wird.

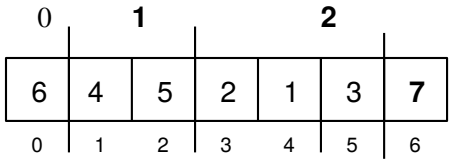
```
//P: ist_sortiert(s[i0:mitte]) und ist_sortiert(s[mitte+1]:i1])
//Q: ist_sortiert(d[i0:i1])
private static void merge(Object[] s, Object[] d,
        int i0, int i1, int mitte, Comparator comp) {
    int l = i0;           // linker Bereich
    int r = mitte + 1;    // rechter Bereich
    // I: ist_sortiert(d[i0:i-1])
         & zu_mischen(l:mitte] & zu_mischen(s[r:i1])
    for (int i = i0; i <= i1; i++) {
        boolean rechtsFertig = r > i1;
        boolean linksWasDa = l <= mitte;
        if (rechtsFertig || linksWasDa && comp.compare(s[l], s[r]) <= 0)
            d[i] = s[l++]
        else
            d[i] = s[r++];
    }
}
```

Vergleich Mergesort und Quicksort

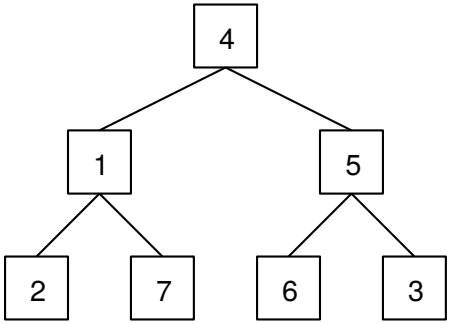
- Mergesort ist garantiert O(n log n), Quicksort im Mittel
- Die Anzahl der Vergleich und der Datenbewegungen ist bei Mergesort $n \log_2(n)$.
- Quicksort hat im Mittel 1.4 $n \log_2 n$ Vergleiche.
- Quicksort hat im Mittel 0.7 $n \log_2 n$ Datenbewegungen.
- Quicksort ist schneller, wenn die Vergleiche schneller sind als die Datenbewegungen (einfache Datentypen)
- Mergesort ist schneller, wenn die Vergleiche die Zeit dominieren (Objekte)
- Es gibt Varianten von Mergesort für Felder und Listen (nicht rekursiv)
- **Mergesort ist stabil** (Beibehalten der Reihenfolge gleichrangiger Elemente).

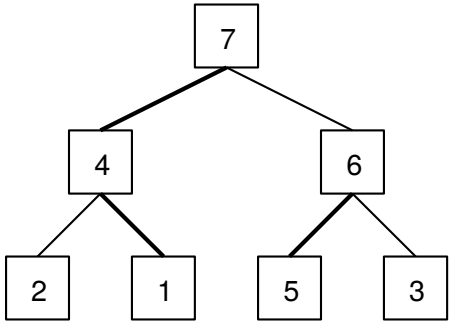
Rekursives Sortieren in einem Baum: Heapsort

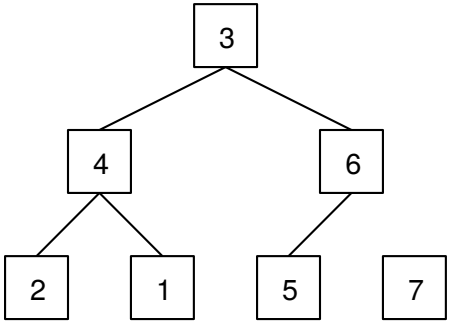


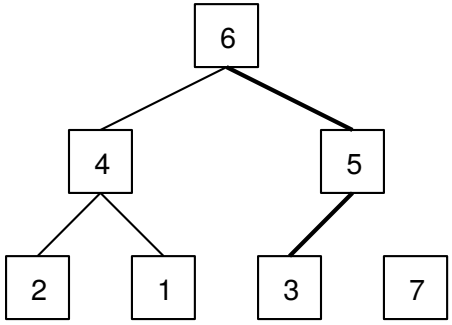


k(i) = 2 * r + i + 1









Heapbedingung: jeder Knoten enthält einen größeren Wert als seine Kinder

1. Feld zu Heap machen

2. Heap von der Spitze her abbauen

Aufwand: O(*n* log *n*)

Berechnungformel Eltern - Kind

Besonders einfach, wenn Array-Indizes 1|2,3|4,5.. zählen:

Kind-Index = 2 * *Eltern-Index* (+ 1 für 2. Kind)
Eltern-Index = *Kind-Index* / 2

Eltern	Kinder
1	2, 3
2	4, 5
3	6, 7
4	8, 9

Bei Indizierung 0|1,2|3,4,5..

Kind-Index = 2 * *Eltern-Index* + 1
Eltern-Index = (*Kind-Index* – 1) / 2

Eltern	Kinder
0	1, 2
1	3, 4
2	5, 6
3	7, 8

Effizientes Einfügen und Entfernen aus einem Heap (O(log *n*))

```
/** Der unterste Knoten schwimmt nach oben, bis er seinen Platz
 *  gefunden hat.
 *  @param child Index des Kindknotens. */
private static void swim(Object[]a, int child, Comparator c) {
    int parent = (child-1) / 2;
    while (child > 0 && c.compare(a[parent], a[child]) < 0) {
        swap(a, parent, child);
        child = parent; parent = (child-1) / 2;
    }
}

/** Der oberste Knoten sinkt nach unten,
 *  bis er seinen Platz gefunden hat.
 *  @param n Größe des Heap. */
private static void sink(Object[] a, int n, Comparator c) {
    int parent = 0, child = 1;
    while (child < n) {
        int sibling = child + 1;
        if (sibling < n && c.compare(a[sibling], a[child]) > 0)
            child = sibling;
        if (c.compare(a[parent], a[child]) >= 0) return;
        swap(a, parent, child);
        parent = child; child = 2 * parent + 1;
    }
}
```

besser: Ringtausch (Feldwerte schieben und dann neuen Wert einsetzen)

Heapsort besteht aus zwei Teilen:
dem Aufbauen und dem Abbauen des Heaps

```
public static void sort(Object[] a, Comparator c) {
    // wir fügen ab dem 2. Feldelement alle Elemente nacheinander
    // dem Heap zu und lassen sie nach oben „schwimmen".
   for (int i= 1; i < a.length; i++)
        swim(a, i, c);

    // wir tauschen das jeweils größte Element an den richtigen Platz
    // und lassen das neue a[0] nach unten „sinken"
   for (int i = a.length - 1; i > 0;  i--) {
       swap(a, 0, i);
       sink(a, i, c);
   }
}
```

Die garantierte Laufzeit O(*n log n*) ist ähnlich gut wie Quicksort/Mergsort

Die Beachtung der Heapeigenschaften ist eine ideale Basis für das fortgesetzte Einfügen und Auswählen des Besten (priority queue)

(vgl. Praktikumsaufgabe:
dabei wird die Heapeigenschaft umgekehrt: Elternelement < Kinder)
Einfügen in queue: *am Ende + swim*
Entfernen aus queue: *am Anfang + Tauschen mit Ende + sink*
)

Proxmap Sort (Idee, bessere Implementierung direkt im Feld)

1. Man bestimmt **Minimum und Maximum** der Datenmenge. O(*n*)
2. Man ermittelt daraus eine **Funktion mapKey()** mit richtiger Reihenfolge!.
3. Mittels mapKey() berechnet man für jedes Datenelement eine **Sollposition** und sortiert die Daten in ein **Bucket** an Sollposition. O(??)
4, Man **überträgt die Daten** der Reihe nach in das Feld. O(*n*)

Kritisch ist vor sind Punkt 2 und 3:

- bei **zufälligen/gleichverteilten** Daten ist der Aufwand insgesamt **O(*n*)** (pro Bucket O(1))
- bei **ungünstigen Daten** (1 Bucket) ist der Aufwand **O(n^2).**
- **schwierig allgemein zu lösen.**

Vergleich der Ausführungszeiten von Sortierverfahren

	64	128	256	512	1024
Quicksort	0,40	0,98	2,22	4,94	10,86
Heapsort*	0,61	1,43	3,28	7,43	16,57
Proxmap	0,38	0,75	1,51	3,00	5,99
Bubblesort	2,76	11,36	46,42	189,35	766,22
Insertionsort	1,12	4,47	17,58	69,89	280,27
Selectionsort	1,40	5,56	22,18	88,66	354,48

Direktes Sortieren: $O(n^2)$

Heapsort, Mergesort $O(n \log n)$

Quicksort: $O_{avg}(n \log n)$ $O_{worst\ case}(n^2)$ der worst case ist statistisch fast unmöglich

Proxmap $O_{avg}(n)$ $O_{worst\ case}(n^2)$ der worst case hängt immer von Daten ab

*Die Messwerte sind zu schlecht. Man kann Heapsort optimieren!