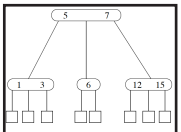
STABIL

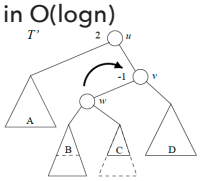
Countsort: Elemente nach Gruppen sortieren z.B. Studierende nach letzter Ziffer von MatrNr (0,1,2,3,4,5,6,7,8,9)
Linear: $\Theta(z + n + n)$

B* Baum in Zwischenknoten nur keys/verweise des >ten Elem. im Unterbaum



B-Baum
Ordnung m
m = 3
KeyPerKid < m
maxKid = m

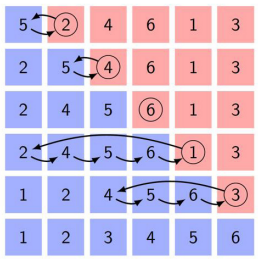
AVL-Baum: - links tiefer
bei >= 2 rebalance.
Einfügen/Entfernen in O(logn)



Rot-Schwarz Baum
wurzel&blatt schwarz
jeder Pfad zu Blatt == #schwarze Knoten

STABIL

Insertionsort



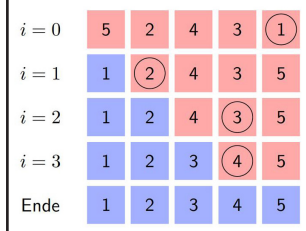
STABIL

Bubblesort:

immer zwei Nachbarn vergleichen,
wenn links größer als rechts -> Tauschen

INSTABIL

Selectionsort



PRIMITIVE

STABIL

Binary Tree Sort
Zeitaufwand für Suchen, Einfügen, Entfernen, Minimum, Maximum, Predecessor und Successor in b.Suchb. mit Höhe h = O(h)
entartet = O(n)
balanciert = (logN)

STABIL

Introsort: C++ aus Ü

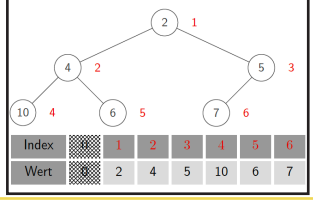
STABIL

TimSort
erst in Runs aufteilen (min-Run), dann wie Mergesort verschmelzen

Min-Gallopping: sobald nicht mehr drin mit binSuche

INSTABIL

Heap Sort
binärer Wurzelbaum
Elter: floor(k/2)



BÄUME

Dijkstra: **verkettete Liste:** Worst-Case-Laufzeit von O(n^2)
priority queue: Q in der Knoten nach dem Wert geordnet; $O((n + m) \log n)$

Gale-Shapely

DIVIDE AND CONQUER

Tabelle: Laufzeit und Vergleiche.

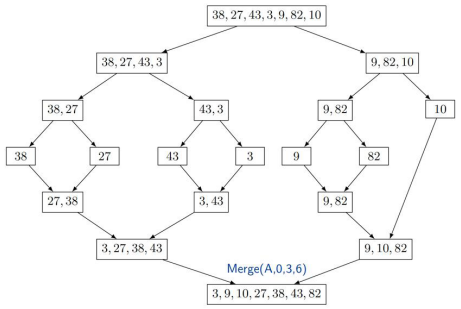
Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabelle: Zusätzlicher Speicher.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

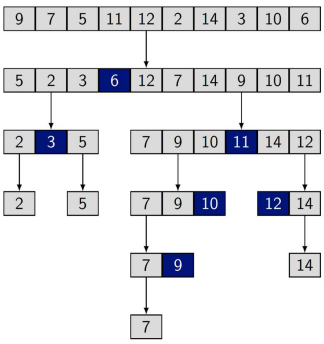
Mergesort: Beispiel

STABIL



Quicksort: Beispiel

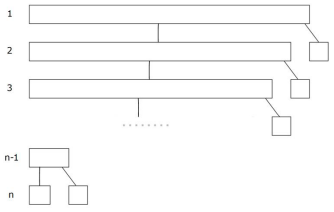
INSTABIL



Best-Case:

- Die beiden Teilfolgen haben immer (ungefähr) die gleiche Länge.
- Die Höhe des Aufrufbaumes ist $\Theta(\log n)$.
- Auf jeder Aufrufebene werden $\Theta(n)$ Vergleiche durchgeführt.
- Die Anzahl der Vergleiche und die Laufzeit liegen in $\Theta(n \log n)$.

Worst-Case: Jede (Teil-)Folge wird immer beim letzten (oder immer beim ersten) Element geteilt.



Die Anzahl der Vergleiche ist $\Theta(n^2)$.

Inversionen zählen

Dichtestes Punktepaar

GREEDY

MST (minimal spanning tree)

Kruskal: repeat:{Kleinste Kante einfügen ohne Kreise zu bilden}
Union Find in konstant
Laufzeit durch Kantensortieren: O(m log n)
licht/dünn: Graph mit Kanten m = ObereSchranke(n) -> Kruskal besser

Prim: Startknoten -> repeat:{Kante mit kleinstem Gewicht das an einem Knoten unseres entdeckten Baums hängt (und keine Kreise)}
Als PrioQ klassischer Heap: O(m logn)
Als PrioQ Fibonacci Heap: O(m+n logn)

dicht: Graph mit Kanten m = scharfeSchranke(n^2) (n..#Knoten) -> Prim besser

Intervall Scheduling O(n log n)
Greedy-Ansatz: Jobs in natürlicher Ordnung: Wähle Job wenn kompatibel (\Überlappung) mit den bisherigen

[Früheste Startzeit] in aufsteigender Reihenfolge von sj
[Früheste Beendigungszeit] in aufst. Reihenfolge von fj
[Kleinstes Intervall] in aufsteigender Reihenfolge von fj − sj
[Wenigste Konflikte] Anzahl cj der nicht kompatiblen Jobs