

**PostOrder**

A->C->B->E->G->F->D

**InOrder**

A->B->C->D->E->F->G

**Binärer Suchbaum**

**Zeitkomplexität Access/Search/Insertion/Delition:** O(log(n)

**Definition/Eigenschaften:**

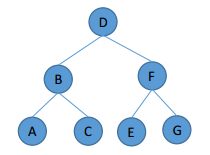
* Knoten bestehen aus Schlüssel-Wert-Paaren
* Knoten unterliegen einer vorgegebenen Ordnung

Blatt: Knoten ohne Kinder

Tiefe eines Knotens: Pfadlänge bis zur Wurzel

**Traversierungsalgorithmen:**   
preOrder: <ROOT> Left Right  
inOder: Left <Root> Right

postOrder: Left Right <Root>



**2-3 Baum**

**Zeitkomplexität Access/Search/Insertion/Delition:** O(log(n)

Ein 2-3-Suchbaum ist entweder leer oder

* ein 2-Kind-Knoten mit einem Schlüssel und 2 Referenzen zu je einem (links: kleineren, rechts: grösseren) 2-3-Suchbaum.
* ein 3-Kind-Knoten mit 2 Schlüsseln und 3 Referenzen zu je einem (links: kleineren, mitte: dazwischenliegenden, rechts: grösseren) 2-3-Suchbaum.

**2-3-Baum:** perfekt balancierter 2-3-Suchbaum = alle null-Referenzen haben gleiche Entfernung zur

Wurzel**Globale Eigenschaften**

* Jede Transformation wahrt die Ordnung und perfekte Balance im ganzen Baum.
* Wenn die Wurzel geteilt wird, erhöht sich die Höhe des Baums um 1.
* 2-3-Bäume wachsen von unten nach oben.
* Höhe eines 2-3-Baumes liegt zwischen ⌊𝑙𝑜𝑔3𝑛⌋ und ⌊𝑙𝑜𝑔2𝑛⌋.

**Preoder**

D->B->A->C->F->E->G

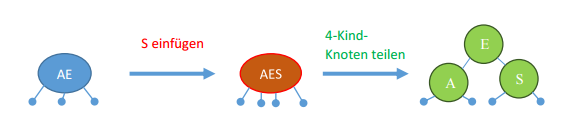
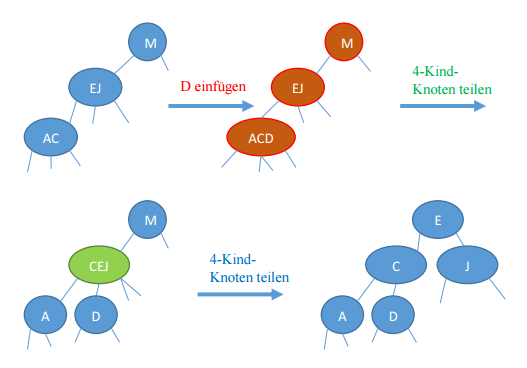
**Balancierte Suchbäume**

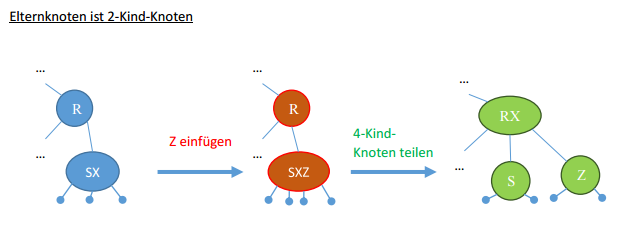
**Bester Fall:** Jeder Elternknoten hat zwei Kinder

**Typischer Fall:** Bäume sind nur teilweise Balanciert

**Schlechtester Fall:** Baum ist einseitig

Daten Zufällig einfügen führt i.d.R zu guten Bäumen. Sortierte Dateneingabe zu schlechten.





**Rot-Schwarz-Bäume**

**Zeitkomplexität Access/Search/Insertion/Delition:** O(log(n)

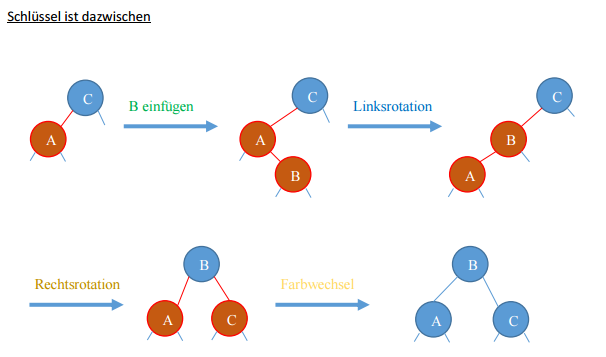
**Definition:** Codiere 2-3-Bäume auf Basis von Standard-Binärsuchbäumen

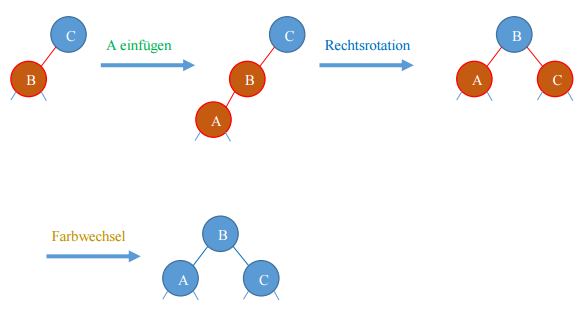
**Rote** Referenzen (bzw. Kindknoten) verbinden 2-Kind-Knoten

**Schwarze** Referenzen (bzw. Kindknoten) verbinden den 2-3-Baum.

**Rote Referenzen zeigen nach links.**

* **Einfügeschema:**   
  Linke Kinder sind Kleiner als Eltern, Rechte Kinder sind grösser als Eltern.
* Beim Einfügen rechts folgt eine links-rotation (eingefügtes Kind wird zum Elternknoten und Elternknoten rutscht nach unten Links. **Kante bleibt Rot**).
* Wenn der Elternknoten & Kinder Balanciert ist, werden die Kanten Schwarz.
* Wenn zwei **aufeinanderfolgende** Kanten Rot sind erfolgt Rechtsrotation & Farbwechsel. Beim Farbwechsel von rot-rot nach schwarz-schwarz wechselt die Farbe des Elternknotens von schwarz auf rot. So werden die roten Referenzen im Baum immer nach oben weitergereicht. Eine rote Wurzel wird abschliessend schwarz gefärbt.





**Hash-Tabellen**

**Zeitkomplexität Access/Search/Insertion/Delition:** O(log(1)

**Idee:** Erweiterung des Prinzips der Speicherung eines Schlüssel-Werte-Paares i im Arrayeintrag i auf

komplizierte Schlüsseltypen.

**Anforderung Hashfunktion:** 1. Konsistent (gleicher Schlüssel muss immer gleicher HashCode haben) **2.** Effizient berechenbar (z.B. String nur erste 30 Zeichen berechnen anstatt alle) **3.** Gleichmässige Verteilung **Einfügen in Hashtabelle:** Position = Rest einer Division der ArrayGrösse. **Bsp**. Zahl: 234287731, Arraygrösse: 1000 (0-999)  Index = 234287731 % 1000 = 731

**Kollisionsauflösung**

**Verkettung:** Jeder der m Arrayindizes enthält eine verkettete Liste mit Schlüsseln, die auf diesen Index

abbilden.

**Lineare Sondierung:** Ist eine Position in der Hashtabelle bereits gefüllt, wird die nächste freie gewählt. Nachteil: Nach jedem herausnemen eines Eintrags muss die HashTabelle neu errechnet werden.

**Insertion Sort**

**Vergleichsoperationen:**

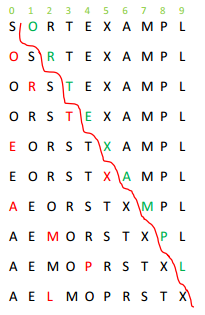
**Best:** n-1 **Avg:** (n2)/4 **Worst:** (n2)/2

**Tauschoperationen:**

**Best:**  **Avg:** (n2)/4 **Worst:** (n2)/2

**Komplexität:** O(n2)

**Pseudocode:** Teilt den Array in einen sortierten und zu-sortierenden Bereich. Fängt links an, nimmt erstes Element aus dem unsortiertem Bereich, vergleicht es mit dem Elementen aus dem Sortierten Bereich bis das erste Element gefunden wird welches grösser ist und stellt es rechts davon in den Array.



**Selection Sort**

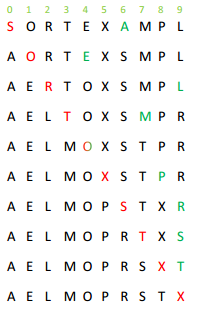
Daten werden im Array direkt sortiert, es wird kein neues Array erstellt (in-place/in-situ).

**Vergleichsoperationen:** n(n-1)/2

**Tauschoperationen:** n

**Komplexität:** O(n2)

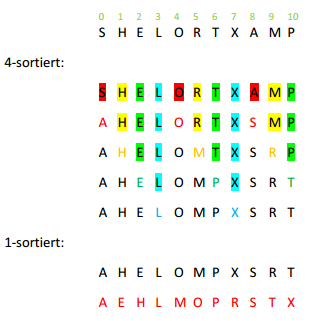
**Pseudocode:** Teilt den Array in einen sortierten und zu-sortierenden Bereich. Fängt links an, geht den zu-sortierenden Bereich durch, nimmt das kleinste Element und stellt es an die letzte Stelle im sortierten Bereich.



**ShellSort**

Sortiere jeweils h-te Elemente mit fester h-Folge

**Komplexität: Best:** O(n) **Avg:** O((nlog(n))2) **Worst:** O((nlog(n))2)



**Insertion Sort**

**Vergleichsoperationen:**

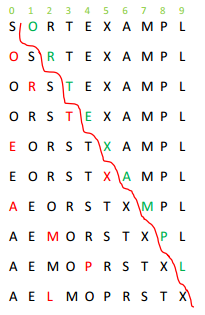
**Best:** n-1 **Avg:** (n2)/4 **Worst:** (n2)/2

**Tauschoperationen:**

**Best:**  **Avg:** (n2)/4 **Worst:** (n2)/2

**Komplexität:** O(n2)

**Pseudocode:** Teilt den Array in einen sortierten und zu-sortierenden Bereich. Fängt links an, nimmt erstes Element aus dem unsortiertem Bereich, vergleicht es mit dem Elementen aus dem Sortierten Bereich bis das erste Element gefunden wird welches grösser ist und stellt es rechts davon in den Array.



**Quick Sort**

**Vergleichsoperationen:**

**Best:** n log n **Avg:** 1.39 n log n **Worst:**  (n2)/2

**Anzahl Arrayzugriffe:** log n (höhe vom Baum)

**Komplexität:** O(n log (n))

**Pseudocode:** Muss im ersten Schritt den Array mischen da er sonst nicht gut funktioniert. Wählt das kleinste Element als Pivot aus und stellt es an die erste Stelle. Wählt das darauf folgende Element als nächsten Pivot aus. Sucht nach rechts laufend das erste Element grösser als der aktuelle Pivot. Dann nach links laufend um das erste kleinere Element vom Pivot zu finden. Wenn zwei Paar gefunden worden sind, vertauscht es diese. Vorgang so lange bis sich die Suchen im Array überkreuzen. Der Pivot wird mit dem letzten kleinerem Element getauscht und der Array an der Stelle vom Pivot in zwei geteilt (pivot bleibt fest). Folge wird für die ausgeführt bis der Array gefüllt ist.

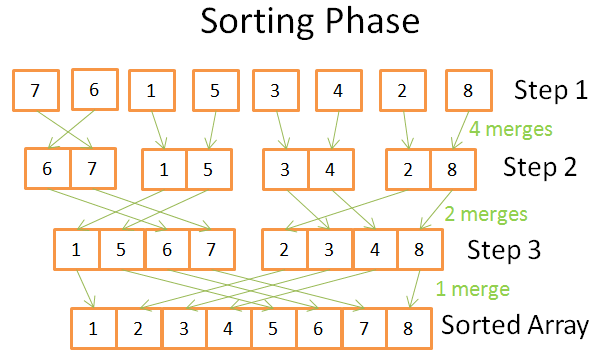


**Merge Sort**

**Vergleichsoperationen:** max 6n log n

**Komplexität:** O(n log (n))

**Pseudocode:** Teilt den Array solange auf bis alle Teile einzeln sind. Vergleicht sie und sortiert sie. Beim *merge* werden die Arrays nochmals verglichen. Das das linke Element wird eingesetzt wenn das damit verglichene rechte Element grösser ist, andernfalls wird das rechte eingesetzt.



**Heap Sort**

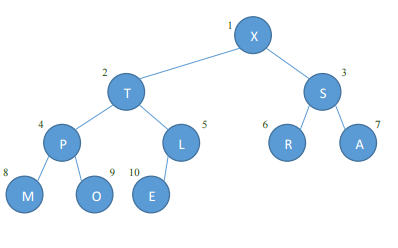
Ein binärer Heap ist in einem vollständigen Heap-geordneten Binärbaum angeordnet, d.h. der Schlüssel jedes Knotens ≤ der Schlüssel seines Elternknotens, und der grösste Schlüssel ist an der Wurzel.

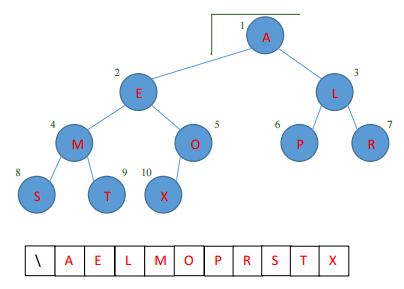
**Vergleichsoperationen:** max. 2𝑛 log 𝑛 + 2𝑛

**Tauschoperationen**: max. 𝑛 log 𝑛 + 𝑛

**Komplexität:** 𝑂(𝑛 log 𝑛)

**Pseudocode:** Wenn der Baum Heap-Sortiert ist (Grösstes Element oben & alle Kinder sind kleiner als ihre Eltern), wird das grösste (Wurzel) mit dem Element unten rechts vertauscht. Das grösste Element der gesamten Datenstruktur ist nun unten rechts. Da jedoch ein kleineres Element nun an der Wurzel ist, muss der Baum wieder geordnet werden um die Heap-Datenstruktur zu bewahren. Hier wird *sink* angewendet. Das Element wird Jeweils mit dem grösseren Kinderknoten getauscht, bis die Heap-Datenstruktur wieder vollkommen ist. Dann wird wieder die Wurzel mit dem Element unten rechts vertauscht usw.





**Rekursion:** Algorithmus oder Datenstruktur die auf sich selbst aufruft

**Baumrekursion:** Pro Schritt mehrere rekursive Aufrufe

**Rekursionsanker:** Direkte Behandlung eincaher, sofort Lösbarer Fälle

**Rekursionsschritt:** Herleitung der Lösung aus der Lösung kleinerer Fälle. Pro Schritt, 1 Aufruf.

**Rekursive Sequenzielle Suche:**

Ist das Feld leer, dann ist das gesuchte Element nicht enthalten (RA 1). Ist das erste Element das gesuchte, dann wurde das Element gefunden (RA 2). Sonst suche im restlichen Feld nach gleichem Verfahren weiter (RS).

**Rekusrive Binäre Suche:**

Ist das Feld leer, dann ist das gesuchte Element nicht enthalten (RA 1). Ist das mittlere das gesuchte, dann wurde das Element gefunden (RA2). Sonst suche entsprechend im linken (<) oder rechten (>) Teil des Feldes nach gleichem Verfahren weiter (RS).

**Bäume**

**Definition:** Datenstruktur bestehend aus Knoten, die über Referenzen/Kanten miteinander verbunden

sind. Auf jeden Knoten darf nur ein anderer Elternknoten zeigen. **Ausnahme:** Wurzel

**Binärbaum**

**Definition:** Von jedem Knoten gehen genau zwei (null-)Referenzen aus: rechte/linke Referenz auf rechtes/linkes

Kind.

**Rekursive Definition:** Ein binärer Baum ist entweder eine null-Referenz oder ein Knoten mit einer

linken und einer rechten Referenz, die jeweils auf disjunkte binäre Teilbäume verweisen.

**Iteratoren**

hasNext(), next(), remove()

Ohne Generics: *Iterator i = collection.iterator()*  
Mit Generics:

*Collection = new List<String> …*

*Iterator<String> I = collection.iterator();*

**Einfach Verkettete Listen**

**Speicherkomplexität**: O(n)

**Laufzeitkomplexitäten**:

**size**: O(1) oder O(n) (entweder als Variable gespeichert oder es geht alle Nodes durch und zählt diese)

**isEmpty/addFirst/getFirst/removeFirst**: O(1)

**addLast/getLast/removeLast**: O(n)

**add/get/remove(int index):** "O(index)"

**Doppelt Verkettete Listen**

**Speicherkomplexität**: O(n)

**Laufzeitkomplexitäten**:

**size**: O(1) oder O(n) (entweder als Variable gespeichert oder es geht alle Nodes durch und zählt diese)

**isEmpty/addFirst/getFirst/removeFirst**: O(1)

**addLast/getLast/removeLast**: O(1)

**add/get/remove(int index):** "O(index)"

**Binary Search:**

**BestCase:** O(1) **WorstCase**: O(log(n) **AverageCase:** O(log(n)

**Voraussetzung:** Daten sind geordnet

**Pseudocode:** Vergleicht gesuchtes Element mit dem Element in der Mitte des Array. Wenn Element kleiner dann verwirft BinarySearch die grössere Hälfte. Wenn es grösser ist dann verwirft es die kleinere Hälfte.

Wird solange durchgeführ bis das Element gefunden ist oder sich die Low und High treffen.

**Sequential Search:**

**BestCase:** O(1) **WorstCase:** O(n) **AverageCase:** O(N)

**Voraussetzung:** Keine

**Pseudocode:** Vergleicht jedes Element mit dem Gesuchten bis er am Ende vom Array ist oder das Element gefunden hat.

* **Que:** First-In-Last-Out  
  addFirst: O(1)  
  addLast: O(n)  
  addAtIndex: O(i)  
  isEmpty: O(1)
* **Stack:** Last-In-First-Out  
  push: O(1)  
  pop: O(1)  
  peek: O(1)  
  isEmpty: O(1)
* **Algorithmus**: Endlich, Deterministisch, Effektive Lösungsverfahren für Computer
* **Datenstrukturen**: Organisationen von Daten und können Anfangs-, Neben- oder Endpunkte von Algorithmen sein
* **Warum ADS:** Durchführung von Aufgaben um Faktoren >1'000'000 beschleunigen, oder deren Lösung überhaupt erst möglich machen. Schnellere Rechner erreicht lediglich eine Verbesserung um Faktoren ca. 10-100.
* **O-Notation:** O(g(n)) ist eine asymptotische obere Schranke für f(n)
* **Abstrakter Datentyp:** Repräsentation bleibt von Client verborgen

|  |  |
| --- | --- |
| **Wachstumsordnung** | **Beschreibung** |
| O(1) | Konstant |
| O(Log(n)) | Logarithmisch |
| O(n) | Linear |
| O(nLog(n)) | Leicht Überlinear |
| n^2 | Quadratisch |
| n^3 | Kubisch |
| 2^n | Exponentiell |