Algorithmen und Datenstrukturen 1

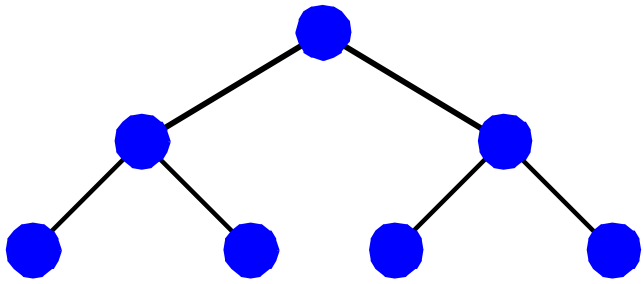
Prof. Dr. E. Rahm

Wintersemester 2001 / 2002

Universität Leipzig

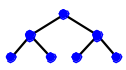
Institut für Informatik

http://dbs.uni-leipzig.de



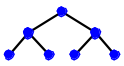
Zur Vorlesung allgemein

- Vorlesungsumfang: 2 + 1 SWS
- Vorlesungsskript
  - im WWW abrufbar (PDF, PS und HTML)
  - Adresse *http://dbs.uni-leipzig.de*
  - ersetzt nicht die Vorlesungsteilnahme !
  - ersetzt nicht zusätzliche Benutzung von Lehrbüchern
- Übungen
  - Durchführung in zweiwöchentlichem Abstand
  - selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
  - Übungsblätter im WWW
  - praktische Übungen auf Basis von Java
  - Rechnerzeiten reserviert im NT-Pool (HG 1-68, Mo-Fr. nachmittags) und Sun-Pool (HG 1-46, vormittags und Mittwoch nachmittags)
  - Detail-Informationen siehe WWW



Leistungsbewertung

- Erwerb des Übungsscheins ADS1 (unbenotet)
  - Fristgerechte Abgabe der Lösungen zu den gestellten Übungsaufgaben
  - Übungsklausur Ende Jan./Anfang Feb.
  - Zulassungsvoraussetzung ist korrekte Lösung der meisten Aufgaben und Bearbeitung aller Übungsblätter (bis auf höchstens eines)
- Informatiker (Diplom), 3. Semester
  - Übungsschein ADS1 zu erwerben (Voraussetzung für Vordiplomsklausur)
  - Klausur über Modul ADS (= ADS1+ADS2) im Juli als Teilprüfung zur Vordiploms-Fachprüfung „Praktische Informatik“
- Mathematiker / Wirtschaftsinformatiker: Übungsschein ADS1 erforderlich
- Magister mit Informatik als 2. Hauptfach
  - kein Übungsschein erforderlich
  - Prüfungsklausur zu ADS1 + ADS2 im Juli
  - Bearbeitung der Übungsaufgaben wird dringend empfohlen

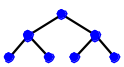


Termine Übungsbetrieb

- Ausgabe 1. Übungsblatt: Montag, 15. 10. 2001; danach 2-wöchentlich
- Abgabe gelöster Übungsaufgaben bis spätestens Montag der übernächsten Woche, 11:15 Uhr
  - vor Hörsaal 13 (Abgabemöglichkeit 11:00 - 11:15 Uhr)
  - oder früher im Holz-Postkasten HG 3. Stock, Abt. Datenbanken
  - Programmieraufgaben: dokumentierte Listings der Quellprogramme sowie Ausführung
- 6 Übungsgruppen

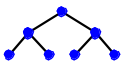
Nr.	Termin	Woche	Hörsaal	**Beginn**	Übungsleiter	#Stud.
1	Mo, 15:15	A	SG 3-11	**29.10**	Sosna	30
2	Mo, 15:15	B	SG 3-11	**5.11**	Sosna	30
3	Di, 11:15	A	SG 3-07	**30.10.**	Böhme	30
4	Di, 11:15	B	SG 3-07	**6.11**	Böhme	30
5	Fr, 15.15	A	HS 20	**2.11**	Müller	60
6	Fr, 15.15	B	HS 20	**9.11**	Müller	60

  - Einschreibung über Online-Formular
  - Aktuelle Infos siehe WWW

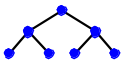


Ansprechpartner ADS1

- Prof. Dr. E. Rahm
  - während/nach der Vorlesung bzw. Sprechstunde (Donn. 14-15 Uhr), HG 3-56
  - rahm@informatik.uni-leipzig.de
- Wissenschaftliche Mitarbeiter
  - Timo Böhme, boehme@informatik.uni-leipzig.de, HG 3-01
  - Robert Müller, mueller@informatik.uni-leipzig.de, HG 3-01
  - Dr. Dieter Sosna, dieter@informatik.uni-leipzig.de, HG 3-04
- Studentische Hilfskräfte
  - Tilo Dietrich, TiloDietrich@gmx.de
  - Katrin Starke, katrin.starke@gmx.de
  - Thomas Tym, mai96iwe@studserv.uni-leipzig.de
- Web-Angelegenheiten:
  - S. Jusek, juseks@informatik.uni-leipzig.de, HG 3-02



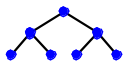
Vorläufiges Inhaltsverzeichnis



Literatur

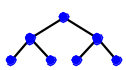
Das intensive Literaturstudium zur Vertiefung der Vorlesung wird dringend empfohlen. Auch Literatur in englischer Sprache sollte verwendet werden.

- *T. Ottmann, P. Widmayer:* Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 3. Auflage, Spektrum-Verlag, 1996
- *M.A. Weiss:* Data Structures & Algorithm Analysis in Java. Addison-Wesley 1999, 2. Auflage 2002
- Weitere Bücher
  - *V. Claus, A. Schwill:* Duden Informatik, BI-Dudenverlag, 2. Auflage 1993
  - *D.A. Knuth:* The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973
  - *R. Sedgewick:* Algorithmen. Addison-Wesley 1992
  - *G. Saake, K. Sattler:* Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 2002
  - *A. Solymosi, U. Gude*: Grundkurs Algorithmen und Datenstrukturen. Eine Einführung in die praktische Informatik mit Java. Vieweg, 2000, 2. Auflage 2001



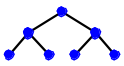
Einführung

- Algorithmen stehen im Mittelpunkt der Informatik
- Wesentliche Entwurfsziele bei Entwicklung von Algorithmen:
  - Korrektheit
  - Terminierung
  - Effizienz
- Wahl der Datenstrukturen v.a. für Effizienz entscheidend
- Abstrakte Datentypen (ADTs): Zusammenfassung von Algorithmen und Datenstrukturen
- Vorlesungsschwerpunkte:
  - Entwurf von effizienten Algorithmen und Datenstrukturen
  - Analyse ihres Verhaltens



Komplexität von Algorithmen

- funktional gleichwertige Algorithmen weisen oft erhebliche Unterschiede in der Effizienz (Komplexität) auf
- Wesentliche Maße:
  - Rechenzeitbedarf (Zeitkomplexität)
  - Speicherplatzbedarf (Speicherplatzkomplexität)
- Programmlaufzeit von zahlreichen Faktoren abhängig
  - Eingabe für das Programm
  - Qualität des vom Compiler generierten Codes und des gebundenen Objektprogramms
  - Leistungsfähigkeit der Maschineninstruktionen, mit deren Hilfe das Programm ausgeführt wird
  - Zeitkomplexität des Algorithmus, der durch das ausgeführte Programm verkörpert wird
- Bestimmung der Komplexität
  - Messungen auf einer bestimmten Maschine
  - Aufwandsbestimmungen für idealisierten Modellrechner (Bsp.: Random-Access-Maschine oder RAM)
  - Abstraktes Komplexitätsmaß zur asymptotischen Kostenschätzung in Abhängigkeit zur Problemgröße (Eingabegröße) n



Bestimmungsfaktoren der Komplexität

- Zeitkomplexität T ist i.a. von "Größe" der Eingabe n abhängig
- Beispiel: $T(n) = a \cdot n^2 + b \cdot n + c$
- Verkleinern der Konstanten b und c

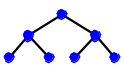
$T_1(n) = n^2 + n + 1$ $\qquad$ $T_2(n) = n^2$

n	1	2	3	10	20	100	1000
$T_1(n)$	3	7	13	111	421	10101	1001001
$T_2(n)$	1	4	9	100	400	10000	1000000
T_1/T_2	3	1.75	1.44	1.11	1.05	1.01	1.001

- Verbessern der Konstanten a nach a'

$$\lim_{n \to \infty} \frac{a \cdot n^2 + b \cdot n + c}{a' \cdot n^2 + b' \cdot n + c'} = \frac{a}{a'}$$

- Wesentlich effektiver: Verbesserung im Funktionsverlauf ! (Wahl eines anderen Algorithmus mit günstigerer Zeitkomplexität)

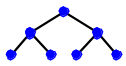


Asymptotische Kostenmaße

- Festlegung der Größenordnung der Komplexität in Abhängigkeit der Eingabegröße: Best Case, Worst Case, Average Case
- Meist Abschätzung oberer Schranken (Worst Case): *Groß-Oh-Notation*
- Zeitkomplexität T(n) eines Algorithmus ist von der Größenordnung n, wenn es Konstanten n_0 und c >0 gibt, so daß für alle Werte von $n > n_0$ gilt

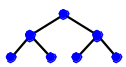
  $T(n) \leq c \cdot n$

  - man sagt "T(n) ist in O(n)" bzw. "$T(n) \in O(n)$" oder "$T(n) = O(n)$"
- Allgemeine Definition:
  Klasse der Funktionen O(f), die zu einer Funktion (Größenordnung) f gehören ist $O(f) = \{g \mid \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
- Ein Programm, dessen Laufzeit oder Speicherplatzbedarf O(f(n)) ist, hat demnach die Wachstumsrate f(n)
  - Beispiel: $f(n) = n^2$ oder $f(n) = n \cdot \log n$.
  - f(n) = O(n log n) -> f(n) = $O(n^2)$, jedoch gilt natürlich O(n log n) ≠ $O(n^2)$



Asymptotische Kostenmaße (2)

- Beispiel: $\mathbf{6n^4 + 3n^3 - 7\ n \in O(n^4)}$
  - zu zeigen: $6n^4 + 3n^3 - 7 <= c\ n^4$ für ein c und alle $n > n_0$
    -> $6 + 3/n - 7 / n^4 <= c$
  - Wähle also z.B. c = 9, $n_0 = 1$
- *Groß-Omega-Notation*: $f \in \Omega(g)$ oder $f = \Omega(g)$ drückt aus, daß f mindestens so stark wächst wie g (untere Schranke)
  - Definition: $\Omega(g) = \{h \mid \exists\ c > 0 : \exists\ n_0 > 0 : \forall\ n >= n_0 : h(n) >= c\ g(n)\}$
  - alternative Definition (u.a. Ottmann/Widmayer):
    $\Omega(g) = \{h \mid \exists\ c > 0 : \exists$ unendlich viele n: $h(n) >= c\ g(n)\}$
- Exakte Schranke: gilt für Funktion f sowohl $f \in O(g)$ als auch $f \in \Omega(g)$, so schreibt man $f = \Theta(g)$
  - *f aus Q(g) bedeutet also*: die Funktion g verläuft ab einem Anfangswert n_0 im Bereich $[c_1 g, c_2 g]$ für geeignete Konstanten c_1, c_2

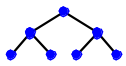


Wichtige Wachstumsfunktionen

- Kostenfunktionen
  - O (1) konstante Kosten
  - O (log n) logarithmisches Wachstum
  - O (n) lineares Wachstum
  - O (n log n) n-log n-Wachstum
  - O (n^2) quadratisches Wachstum
  - O (n^3) kubisches Wachstum
  - O (2^n) exponentielles Wachstum

- Wachstumsverhalten

log n	3	7	10	13	17	20
$\sqrt{n}$	3	10	30	100	300	1000
n	**10**	**100**	**1000**	**10^4**	**10^5**	**10^6**
n log n	30	700	10^4	10^5	$2 \cdot 10^6$	$2 \cdot 10^7$
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1000	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

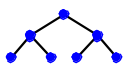


- Problemgröße bei vorgegebener Zeit

Komplexität	1 sec	1 min	1 h
$\log_2 n$	2^{1000}	2^{60000}	-
n	**1000**	**60000**	**3600000**
n $\log_2 n$	140	4893	20000
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

- Größe des größten Problems, das in 1 Stunde gelöst werden kann:

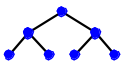
Problemkomplexität	aktuelle Rechner	Rechner 100x schneller	1000x schneller
n	N_1	100 N_1	1000 N_1
n^2	N_1	10 N_2	32 N_2
n^3	N_3	4.6 N_3	10 N_3
n^5	N_4	2.5 N_4	4 N_4
2^n	N_5	N_1 + 7	N_1 + 10
3^n	N_6	N_6 + 4	N_6 + 6



Leistungsverhalten bei kleiner Eingabegröße

- Asymptotische Komplexität gilt vor allem für große n
- bei kleineren Probleme haben konstante Parameter wesentliche Einfluß
- Verfahren mit besserer (asympt.) Komplexität kann schlechter abschneiden als Verfahren mit schlechter Komplexität

Alg.	T(n)	Bereiche von n mit günstigster Zeitkomplexität
A_1	$186182 \log_2 n$	$n > 2048$
A_2	$1000\ n$	$1024 \leq n \leq 2048$
A_3	$100\ n \log_2 n$	$59 \leq n \leq 1024$
A_4	$10\ n^2$	$10 \leq n \leq 58$
A_5	n^3	$n = 10$
A_6	2^n	$2 \leq n \leq 9$



Zeitkomplexitätsklassen

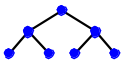
- Drei zentrale *Zeitkomplexitätsklassen* werden unterschieden
- Algorithmus A mit Zeitkomplexität T(n) heißt:

linear-zeitbeschränkt $\mathbf{T(n) \in O(n)}$

polynomial-zeitbeschränkt $\mathbf{\exists_k \in N}$, so daß $\mathbf{T(n) \in O(n^k)}$

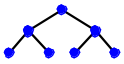
exponentiell-zeitbeschränkt $\mathbf{\exists_k \in N}$, so daß $\mathbf{T(n) \in O(k^n)}$

- exponentiell-zeitbeschränkte Algorizhmen im allgemeinen (größere n) nicht nutzbar
- Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, gelten als unlösbar (intractable)



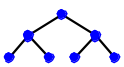
Berechnung der (Worst-Case-) Zeitkomplexität

- elementare Operationen (Zuweisung, Ein-/Ausgabe): *O (1)*
- Summenregel:
  - $T_1(n)$ und $T_2(n)$ seien die Laufzeiten zweier Programmfragmente P_1 und P_2; es gelte $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$.
  - Für die Hintereinanderausführung von P_1 und P_2 ist dann
    $$T_1(n) + T_2(n) \in O(max(f(n), g(n)))$$
- Produktregel, z.B. für geschachtelte Schleifenausführung von P_1 und P_2:
  $$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n))$$
- Weitere Konstrukte
  - Fallunterscheidung: Kosten der Bedingungsanweisung (= O(1)) + Kosten der längsten Alternative
  - Schleife: Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung
  - rekursive Prozeduraufrufe: Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung



Beispiel zur Bestimmung der Zeitkomplexität

```
void proz0 (int n) {
        proz1();
        proz1();
        for (int i=1; i <= n; i++) {
                proz2();
                proz2();
                proz2();
                for (int j=1; j <= n; j++) {
                        proz3();
                        proz3();
                }
        }

}
```

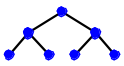
Beispiel: Berechnung der maximalen Teilsumme

- Gegeben: Folge F von n ganzen Zahlen. Gesucht: Teilfolge von 0 <= i <= n aufeinander folgenden Zahlen in F, deren Summe maximal ist
- Anwendungsbeispiel: Entwicklung von Aktienkursen (tägliche Änderung des Kurses). Maximale Teilsumme bestimmt optimales Ergebnis

Tag	1	2	3	4	5	6	7	8	9	10
Gewinn/Verlust (Folge)	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

- Lösungsmöglichkeit 1:

```
int maxSubSum1( int [ ] a) {
      int maxSum = 0; // leere Folge
      for ( int i = 0; i < a.length; i++ )
         for ( int j = i; j < a.length; j++ ) {
            int thisSum = 0;
            for ( int k = i; k <= j; k++ )
               thisSum += a[ k ];
            if ( thisSum > maxSum ) maxSum = thisSum;
         }
      return maxSum;
   }
```

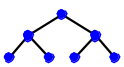



Komplexitätsbestimmung

- Anzahl Durchläufe der äussersten Schleife: n
- mittlere Schleife: berücksichtigt alle Teilfolgen beginnend ab Position i
  - Anzahl: n, n-1, n-2, ... 1
  - Mittel: n+1/2
- Anzahl Teilfolgen: $\sum i = (n^2+n)/2$
- innerste Schleife: Addition aller Werte pro Teilfolge
- #Additionen: $\sum i(n+1-i) = \sum i\, n + \sum i - \sum i^2$

  $= n\,(n^2+n)/2 + (n^2+n)/2 - n/6\,(n+1)\,(2n+1)$

  $= n^3/6 + n^2/2 + n/3$
- Zeitkomplexität:

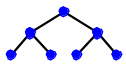


Maximale Teilsumme (2)

- Lösungsmöglichkeit 2:

```
int maxSubSum2 (int [ ] a ) {
      int maxSum = 0; // leere Folge
      for (int i = 0; i < a.length; i++ ) {
         int thisSum = 0;
         for ( int j = i; j < a.length; j++ ) {
            thisSum += a[ j ];
            if (thisSum > maxSum ) maxSum = thisSum;
         }
      }
      return maxSum;
}
```

- Zeitkomplexität:

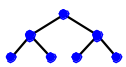


Maximale Teilsumme (3)

- Lösungsmöglichkeit 3:

```
int maxSubSum3 ( int [ ] a ) {
      int maxSum = 0;
      int thisSum = 0;
      for( int i = 0, j = 0; j < a.length; j++ ) {
         thisSum += a[ j ];
         if( thisSum > maxSum ) maxSum = thisSum;
         else if( thisSum < 0 ) {
            i = j + 1;
            thisSum = 0;
         }
      }
      return maxSum;
   }
```

- Zeitkomplexität:
- gibt es Lösungen mit besserer Komplexität?



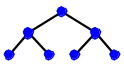
Rekursion vs. Iteration

- für viele Probleme gibt es sowohl rekursive als auch iterative Lösungsmöglichkeiten
- Unterschiede bezüglich
  - Einfachheit, Verständlichkeit
  - Zeitkomplexität
  - Speicherkomplexität
- Beispiel: Berechnung der Fakultät n!

```
int fakRekursiv (int n) { // erfordert n > 0
        if (n <= 1) return 1;
        else return n * fakRekursiv (n-1);
}
```

```
int fakIterativ (int n) { // erfordert n > 0
    int fak = 1;
    for (int i = 2; i <= n; i++) fak *= i;
    return fak;
}
```

  - Zeitkomplexität
  - Speicherkomplexität

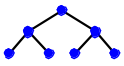


Berechnung der Fibonacci-Zahlen

- Definition
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$
- rekursive Lösung verursacht exponentiellen Aufwand
- iterative Lösung mit linearem Aufwand möglich
  - z.B. Speichern der bereits berechneten Fibonacci-Zahlen in Array
  - Alternative *fibIterativ*

```
int fibRekursiv (int n) { // erfordert n > 0
        if (n <= 0) return 0;
        else if (n ==1) return 1;
        else return fibRekursiv (n-2) + fibRekursiv (n-1);
}
```

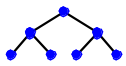
```
int fibIterativ (int n) { // erfordert n > 0
        if (n <=0) return 0;
        else {
        int aktuelle =1, vorherige = 0, temp =1,
        for (int i = 1; i<n; i++) {
                temp=aktuelle;
                aktuelle += vorherige;
                vorherige = temp; }
        return aktuelle;
}
```

Das Prinzip "Teile und Herrsche" (Divide and Conquer)

- Komplexität eines Algorithmus läßt sich vielfach durch Zerlegung in kleinere Teilprobleme verbessern
- Lösungsschema

1. *Divide:* Teile das Problem der Größe n in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn n > 1 ist; sonst löse das Problem der Größe 1 direkt.
2. *Conquer:* Löse die Teilprobleme auf dieselbe Art (rekursiv).
3. *Merge:* Füge die Teillösungen zur Gesamtlösung zusammen.



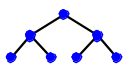
Beispiel: Sortieren einer Liste mit n Elementen

- einfache Sortierverfahren: O (n^2)
- Divide-and-Conquer-Strategie:

```
Modul     Sortiere (Liste)(* Sortiert Liste von n Elementen *)
Falls n > 1 dann
     Sortiere (erste Listenhälfte)
     Sortiere (zweite Listenhälfte)
     Mische beide Hälften zusammen.
```

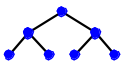
- Kosten $T(n) = 2 \cdot T(n/2) + c \cdot n \qquad T(1) = d$
- Diese rekursives Gleichungssystem (Rekurrenzrelation) hat geschlossene Lösung $T(n) = c \cdot n \cdot \log_2 n + d \cdot n$

=> Sortieralgorithmus in *O(n log n)*



Beispiel 2: Maximale Teilsumme

- rechtes Randmaximum einer Folge
  - rechte Randfolge von F = Teilfolge von F, die bis zum rechten Rand (Ende) von F reicht
  - rechtes Randmaximum von F: maximale Summe aller rechten Randfolgen
  - analog: linke Randfolge, linkes Randmaximum
- Beispiel: F = (+3, -2, +5, -20, +3, +3)
- rekursiver (Divide-and-Conquer-) Algorithmus für maximale Teilsumme
  - falls Eingabefolge F nur aus einer Zahl *z* besteht, nimm Maximum von *z* und 0
  - falls F wenigstens 2 Elemente umfasst:
    - zerlege F in etwa zwei gleich große Hälften *links* und *rechts*
    - bestimme maximale Teilsumme, *ml*, sowie rechtes Randmaximum, *rR*, von *links*
    - bestimme maximale Teilsumme, *mr*, sowie linkes Randmaximum, *lR*, von *rechts*
    - das Maximum der drei Zahlen *ml*, *rR*+*lR*, und *mr* ist die maximale Teilsumme von F



Multiplikation zweier n-stelliger Zahlen

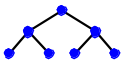
- Standardverfahren aus der Schule: $O(n^2)$

```
 5432 • 1995
-------------
   5432
  48888
   48888
    27160
-------------
 10836840
```

- Verbesserung: Rückführung auf Multiplikation von 2-stelligen Zahlen

A	B		C	D
54	32	•	19	95

$$AC = 54 \cdot 19 = 1026$$
$$(A + B) \cdot (C + D) - AC - BD = 86 \cdot 114 - 1026 - 3040 = 5738$$
$$BD = 32 \cdot 95 = 3040$$
$$10836840$$

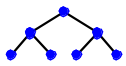


Multiplikation (2)

- Prinzip auf n-stellige Zahlen verallgemeinerbar
- Kosten
  - drei Multiplikationen von Zahlen mit halber Länge
  - Aufwand für Addition und Subtraktion proportional zu n:

    $T(n) = 3T(n/2) + c \cdot n \qquad T(1) = d$
  - Die Lösung der Rekurrenzrelation ergibt sich zu

    $T(n) = (2c + d)n^{\log 3} - n$.
  - Kosten proportional zu $n^{\log 3}$ ($O(n^{1.59})$)

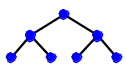


Problemkomplexität

- Komplexität eines Problems: Komplexität des besten Algorithmus'
- Aufwand typischer Problemklassen

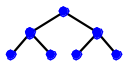
Komplexität	*Beispiele*
O(1)	einige Suchverfahren (Hashing)
O(log n)	allgemeinere Suchverfahren (Binärsuche, Baum-Suchverfahren)
O(n)	sequentielle Suche, Suche in Texten; maximale Teilsumme einer Folge, Fakultät, Fibonacci-Zahlen
O(n log n)	Sortieren
$O(n^2)$	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$	viele Optimierungsprobleme, Türme von Hanoi, Acht-Damen-Problem

- theoretisch nicht lösbare algorithmische Probleme: Halteproblem, Gleichwertigkeit von Algorithmen
- nicht-algorithmische Probleme



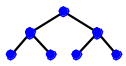
Zusammenfassung

- Komplexität / Effizienz wesentliche Eigenschaft von Algorithmen
- meist asymptotische Worst-Case-Abschätzung in Bezug auf Problemgröße n
  - Unabhängigkeit von konkreten Umgebungsparametern (Hardware, Betriebsystem, ...)
  - asymptotisch „schlechte“ Verfahren können bei kleiner Problemgröße ausreichen
- wichtige Klassen: O(1), O(log n), O (n), O (n log n), O (n^2), ... O(2^n)
- zu gegebener Problemstellung gibt es oft Algorithmen mit stark unterschiedlicher Komplexität
  - unterschiedliche Lösungsstrategien
  - Raum vs. Zeit: Zwischenspeichern von Ergebnissen statt mehrfacher Berechnung
  - Iteration vs. Rekursion
- Bestimmung der Komplexität aus Programmfragmenten
- allgemeine Lösungsstrategie: Divide-and-Conquer (Teile und Herrsche)



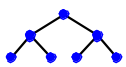
2. Einfache Suchverfahren

- Lineare Listen
- Sequentielle Suche
- Binäre Suche
- Weitere Suchverfahren auf sortierten Feldern
  - Fibonacci-Suche
  - Sprungsuche
  - Exponentielle Suche
  - Interpolationssuche
- Auswahlproblem



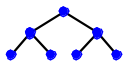
Lineare Listen

- Lineare Liste
  - endliche Folge von Elementen eines Grundtyps (Elementtyps)
    $<a_1, a_2, \ldots a_n>$ $n >= 0$
    n=0: leere Liste < >
  - Position von Elementen in der Liste ist wesentlich
- Typische Operationen
  - INIT (L): Initialisiert L, d.h. L wird eine leere Liste
  - INSERT (L, x, p): Fügt x an Position p in Liste L ein und verschiebt die Elemente an p und den nachfolgenden Positionen auf die jeweils nächsthöhere Position
  - DELETE (L, p): Löscht das Element an Position p der Liste L
  - ISEMPTY (L): Ermittelt, ob Liste L leer ist
  - SEARCH (L, x) bzw. LOCATE (L, x): Prüft, ob x in L vorkommt bzw. gibt die erste Position von L, in der x vorkommt, zurück
  - RETRIEVE (L, p): Liefert das Element an Position p der Liste L zurück
  - FIRST (L): Liefert die erste Position der Liste zurück
  - NEXT (p, L) und PREVIOUS (p, L): Liefert Element der Liste, das Position p nachfolgt bzw. vorausgeht, zurück
  - PRINTLIST (L): Schreibt die Elemente der Liste L in der Reihenfolge ihres Auftretens aus
  - CONCAT (L1, L2): Hintereinanderfügen von L1 und L2



Lineare Listen (2)

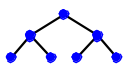
- Komplexität der Operationen abhängig von
  - gewählter Implementierung sowie
  - ob Sortierung der Elemente vorliegt
- Wesentliche Implementierungsalternativen
  - Sequentielle Speicherung (Reihung, Array)
  - Verkettete Speicherung
- Sequentielle Speicherung (Reihung, Array)
  - statische Datenstruktur (hoher Speicheraufwand)
  - wahlfreie Zugriffsmöglichkeit über (berechenbaren) Index
  - 1-dimensionale vs. mehrdimensionale Arrays
- Verkettete Speicherung
  - dynamische Datenstruktur
  - sequentielle Navigation
  - Varianten: einfache vs. doppelte Verkettung etc.



Beispiel-Spezifikation in Java

- Nutzung des Interface-Konzepts (ADT-Umsetzung)
  - Interface: Sammlung abstrakter Methoden
  - Implementierung erfolgt durch Klassen (instanziierbar)
  - Klassen können mehrere Interfaces implementieren (Simulation der Mehrfachvererbung)
  - mehrere Implementierungen pro Interface möglich

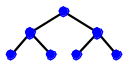
```
public interface Liste { // Annahme Schlüssel vom Typ int
   public void insert (int x, int p) throws ListException;
   public void delete (int p) throws ListException;
   public boolean isempty ();
   public boolean search (int x);
   ...
   public Liste concat (Liste L1, Liste L2);
}

public class ListException extends RuntimeException {
       public ListException (String msg) {super (msg); }
       public ListException () { }
}
```

Beispiel (2)

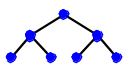
- Einsatzbeispiel für Interface Liste
  - Implementierung ArrayListe wird vorausgesetzt

```
public class ListExample {
        public static void main (String args[]) {
         Liste list = new ArrayListe ();

        try {
         for ( int i = 1; i <= 20; i++ ) {
                   list.insert (i*i,i);
                   System.out.println (i*i);
                   }
        // ...
         if (! list.search (1000)) list.insert (1000,1);
        }
        catch (ListException exc) {
                   System.out.println (exc);
                   }
        }
}
```

Array-Realisierung linearer Listen

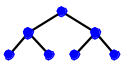
```
public class ArrayListe implements Liste {

    int[] L = null; // Spezialfall: Elemente vom Typ int
    int laenge=0;
    int maxLaenge;

  /* Konstruktoren */

  public ArrayListe(int max) {
       L = new int [max+1]; // Feldindex 0 bleibt reserviert
       maxLaenge = max;
  }
  public ArrayListe() { // Default-Größe 100
       L = new int [101]; // Feldindex 0 bleibt reserviert
       maxLaenge = 100;
  }
```

Array-Realisierung (2)

```
public void insert (int x, int pos) throws ListException {
// Einfügen an Positon pos (Löschen analog)
  if (laenge == maxLaenge) throw new ListException("Liste voll!");
   else if ((pos < 1) || (pos > laenge + 1))
        throw new ListException("Falsche Position!");
   else {
        for (int i=laenge; i >= pos; i--) L[i+1] = L[i];
        L [pos] = x;
        laenge++;
   }
}
// weitere Methoden von ArrayListe
}
```

- Einfüge-Komplexität für Liste mit n Elementen:



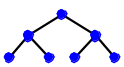
Sequentielle Suche

- Suche nach Element mit Schlüsselwert x
- Falls unbekannt, ob die Elemente der Liste nach ihren Schlüsselwerten sortiert sind, ist die Liste sequentiell zu durchlaufen und elementweise zu überprüfen (sequentielle Suche)

```
public boolean search (int x) {
        // Annahme: Methode eingebettet in Klasse ArrayListe
   int pos = 1;
   while ((pos <= laenge) && (L [pos] != x)) pos++;
   return (pos <= laenge);
}
```

- Kosten
  - erfolglose Suche erfordert n Schleifendurchläufe
  - erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche (und n-1 Schleifendurchläufe)
  - mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche:

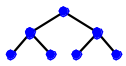
$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{n+1}{2}$$



Sequentielle Suche (2)

- leichte Verbesserung: vereinfachte Schleifenbedingung durch Hinzufügen eines Stoppers bzw. Wächters (englisch “Sentinel”) in Position 0 der Liste

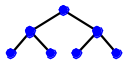
```
public boolean searchSeqStopper(int x) {
  int pos = laenge;
  L[0] = x;
  while (L[pos] != x) pos--;
  return (pos > 0);
}
```

Binäre Suche

- auf sortierten Listen können Suchvorgänge effizienter durchgeführt werden
  - sequentielle Suche auf sortierten Listen bringt nur geringe Verbesserungen (für erfolglose Suche)
  - Binärsuche wesentlich effizienter durch Einsatz der ***Divide-and-Conquer-Strategie***
- Suche nach Schlüssel x in Liste mit aufsteigend sortierten Schlüsseln:
  - Falls Liste leer ist, endet die Suche erfolglos.
    Sonst: Betrachte Element L [m] an mittlerer Position m
  - Falls x = L [m] dann ist das gesuchte Element gefunden.
  - Falls x < L [m], dann durchsuche die linke Teilliste von Position 1 bis m-1 nach demselben Verfahren
  - Sonst (x > L [m]) durchsuche die rechte Teilliste von Position m+1 bis Laenge nach demselben Verfahren
- Beispiel

  Liste: 3 8 17 22 30 32 36 42 43 49 53 55 61 66 75



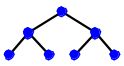
Binäre Suche (2)

- Iterative Lösung

```
public boolean binarySearch(int x) {
  int pos;    /* aktuelle Suchposition
                 (enthaelt bei erfolgreicher Suche Ergebnis) */
  int ug = 1; // Untergrenze des aktuellen Suchbereichs
  int og = laenge; // Obergrenze des aktuellen Suchbereichs
  boolean gefunden = false;
  while ((ug <= og) && (! gefunden)) {
    pos = (ug + og) / 2;
    if (L[pos] > x)
       og = pos - 1; // linken Bereich durchsuchen
    else if (L[pos] < x)
       ug = pos + 1; // rechten Bereich durchsuchen
    else
      gefunden = true;
  }
  return gefunden;
}
```

- Kosten

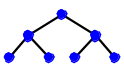
$C_{min}(n) = 1$ $C_{max}(n) = \lceil \log_2(n+1) \rceil$ $C_{avg}(n) \approx \log_2(n+1) - 1$, für große n



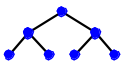
Suche auf Objekten

- bisher Suche nach int-Schlüsseln
- Verallgemeinerung auf Schlüssel (Objekte), auf denen Ordnung vorliegt und zwischen denen Vergleiche möglich sind

```
public interface Orderable {
    public boolean equals (Orderable o);
    public boolean less (Orderable o);
    public boolean greater (Orderable o);
    public boolean lessEqual (Orderable o);
    public boolean greaterEqual (Orderable o);
    public Orderable minKey ();
}
public class OrderableFloat implements Orderable {
    float key; // Schlüssel vom Typ float
    String wert; // weitere Inhalte
    OrderableFloat (float f) {this.key=f;} // Konstruktor
    public boolean equals (Orderable o) {
        return this.key==((OrderableFloat)o).key;}
    public boolean less (Orderable o) {
        return this.key < ((OrderableFloat)o).key;}
    ...
    public Orderable minKey () {
        return new OrderableFloat(Float.MIN_VALUE);}
}
```

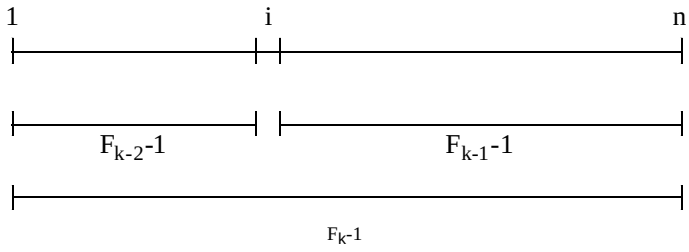



```
public class ArrayListe2 implements Liste2 {
// Liste2 entspricht Liste, jedoch mit Orderable- statt int-Elemeneten
        Orderable[] L = null;
        int laenge=0;
        int maxLaenge;

  public ArrayListe2(int max) {
        L = new Orderable [max+1];
        maxLaenge = max;
  }

public boolean search (Orderable x) {
  int pos = 1;
  while ((pos <= laenge) && (! L [pos].equals (x))) pos++;
  return (pos <= laenge);
}

// weitere Operationen: binarySearch etc. ...
}
```

Fibonacci-Suche

- ähnlich der Binärsuche, jedoch wird Suchbereich entsprechend der Folge der Fibonacci-Zahlen geteilt
- Teilung einer Liste mit n = F_k-1 sortierten Elementen:



Def. Fibonacci-Zahlen:.

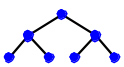
$F_0 = 0$,

$F_1 = 1$,

$F_k = F_{k-1} + F_{k-2}$ für $k \geq 2$

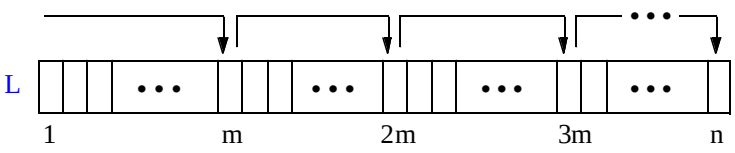
- Element an der Position i = F_{k-2} wird mit dem Suchschlüssel x verglichen
- Wird Gleichheit festgestellt, endet die Suche erfolgreich.
- Ist x größer, wird der rechte Bereich mit F_{k-1}-1 Elementen, ansonsten der linke Bereich mit F_{k-2}-1 Elementen auf dieselbe Weise durchsucht.

- Kosten
  - für n= F_k-1 sind im schlechtesten Fall k-2 Suchschritte notwendig, d.h. O(k) Schlüsselvergleiche
  - Da gilt $F_k \approx c \cdot 1{,}618^k$, folgt $C_{max}(n) = O(\log_{1.618}(n+1)) = O(\log n)$.



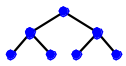
Sprungsuche

- Prinzip
  - der sortierte Datenbestand wird zunächst in Sprüngen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält



  - danach wird der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht
- Einfache Sprungsuche
  - konstante Sprünge zu Positionen m, 2m, 3m ...
  - Sobald `x <= L[i]` mit `i=j*m (j=1,2,...)` wird im Abschnitt `L[(j-1)m+1]` bis `L[j*m]` sequentiell nach dem Suchschlüssel x gesucht.
- mittlere Suchkosten

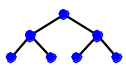
  ein Sprung koste a; ein sequentieller Vergleich b Einheiten $C_{avg}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m - 1)$
- optimale Sprungweite: $m = \sqrt{(a/b)n}$ bzw. $m = \sqrt{n}$
  - falls a=b => $C_{avg}(n) = a\sqrt{n} - a/2$
- Komplexität $O(\sqrt{n})$



Sprungsuche (2)

- Zwei-Ebenen-Sprungsuche
  - statt sequentieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequentiell gesucht wird
  - Mittlere Kosten: $C_{avg}(n) \le \frac{1}{2} \cdot a \cdot \sqrt{n} + \frac{1}{2} \cdot b \cdot n^{\frac{1}{4}} + \frac{1}{2} \cdot c \cdot n^{\frac{1}{4}}$

    a Kosten eines Sprungs auf der ersten Ebene;
    b Kosten eines Sprungs auf der zweiten Ebene;
    c Kosten für einen sequentiellen Vergleich
- Verbesserung durch optimale Abstimmung der Sprungweiten m_1 und m_2 der beiden Ebenen
  - Mit a = b = c ergeben sich als optimale Sprungweiten $m_1 = n^{\frac{2}{3}}$ und $m_2 = n^{\frac{1}{3}}$
  - mittlere Suchkosten: $C_{avg}(n) = \frac{3}{2} \cdot a \cdot n^{\frac{1}{3}}$
- Verallgemeinerung zu n-Ebenen-Verfahren ergibt ähnlich günstige Kosten wie Binärsuche (Übereinstimmung bei $\log_2 n$ Ebenen)
- Sprungsuche vorteilhaft, wenn Binärsuche nicht anwendbar ist (z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher)

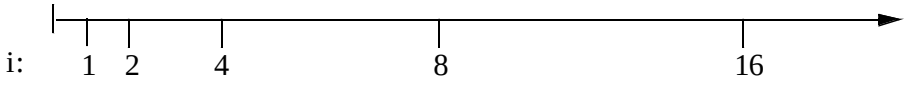


Exponentielle Suche

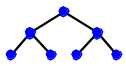
- Anwendung, wenn Länge des Suchbereichs n zunächst unbekannt bzw. sehr groß
- Vorgehensweise
  - für Suchschlüssel x wird zunächst obere Grenze für den zu durchsuchenden Abschnitt bestimmt

```
int i = 1; while (x > L[i]) i=2*i;
```

  - Für i > 1 gilt für den auf diese Weise bestimmten Suchabschnitt: `L[i/2] < x <= L[i]`
  - Suche innerhalb des Abschnitts mit irgendeinem Verfahren



- enthält die sortierte Liste nur positive, ganzzahlige Schlüssel ohne Duplikate, wachsen Schlüsselwerte mindestens so stark wie die Indizes der Elemente
  => i wird höchstens $\log_2 x$ mal verdoppelt
  - Bestimmung des gesuchten Intervalls erfordert maximal $\log_2 x$ Schlüsselvergleiche
  - Suche innerhalb des Abschnitts (z.B. mit Binärsuche) erfordert auch höchstens $\log_2 x$ Schlüsselvergleiche
- Gesamtaufwand $O(\log_2 x)$

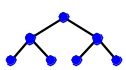


Interpolationssuche

- Schnellere Lokalisierung des Suchbereichs in dem Schlüsselwerte selbst betrachtet werden, um “Abstand” zum Suchschlüssel x abzuschätzen
- nächste Suchposition *pos* wird aus den Werten *ug* und *og* der Unter- und Obergrenze des aktuellen Suchbereichs wie folgt berechnet:

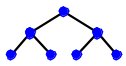
$$pos = ug + \frac{x - L[ug]}{L[og] - L[ug]} \cdot (og - ug)$$

- sinnvoll, wenn Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt
  - erfordert dann im Mittel lediglich $log_2\ log_2\ n + 1$ Schlüsselvergleiche
- im schlechtesten Fall (stark ungleichmäßige Werteverteilung) entsteht jedoch linearer Suchaufwand (O(n))



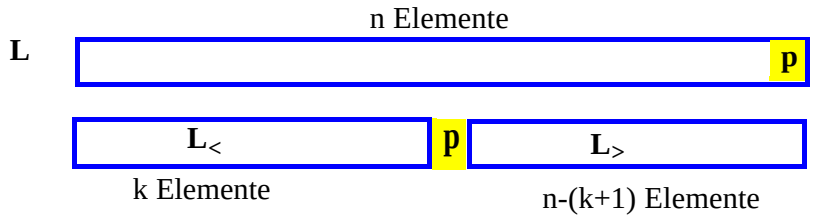
Auswahlproblem

- Finde das i-kleinste Element in einer Liste L mit n Elementen
  - Spezialfälle: kleinster Wert, größter Wert, mittlerer Wert (Median)
- trivial bei sortierter Liste
- unsortierter Liste: Minimum/Maximum-Bestimmung erfordert lineare Kosten O (n)
- einfache Lösung für i-kleinstes Element:
  - j = 1
  - solange j < i: Bestimme kleinstes Element in L und entferne es aus L ; erhöhe j um 1
  - gebe Minimum der Restliste als Ergebnis zurück
- Komplexität:
- schneller: Sortieren + Auswahl

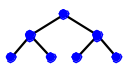


Auswahlproblem (2)

- Divide-and-Conquer-Strategie (i-kleinstes Element von n paarweise unterschiedlichen Elementen)
  - bestimme Pivot-Element p
  - Teile die n Elemente bezüglich p in 2 Gruppen: Gruppe 1 enthält die k Elemente die kleiner sind als p; Gruppe 2 die n-k-1 Elemente, die größer als p sind
  - falls i=k+1, dann ist p das Ergebnis.
    falls i <= k wende das Verfahren rekursiv auf Gruppe 1 an;
    falls i > k+1: verwende Verfahren rekursiv zur Bestimmung des i - (k+1)-te Element in Gruppe 2

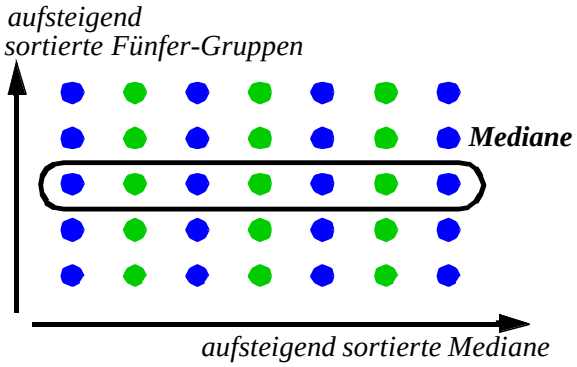


- Laufzeit:

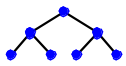


Auswahlproblem (3)

- Lösung des Auswahlproblems mit linearen Kosten: Median-of-Median-Strategie:
  - falls n < Konstante, berechne i-kleinstes Element direkt und beende Algorithmus Sonst:
  - teile die n Elemente in n/5 Gruppen zu je 5 Elementen und höchstens eine Gruppe mit höchstens vier Elemente auf) -> *lineare Kosten*
  - sortiere jede dieser Gruppen (in konstanter Zeit) und bestimme in jeder Gruppe das mittlere Element (Median) -> *lineare Kosten*

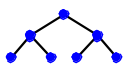


  - wende Verfahren rekursiv auf die Mediane an und bestimme das mittlere Element p (Median der Mediane) -> Pivot-Element
  - Teile die n Elemente bezüglich p in 2 Gruppen: Gruppe 1 enthält die k Elemente die kleiner sind als p; Gruppe 2 die n-k-1 Elemente, die größer als p sind
  - falls i=k+1, dann ist p das Ergebnis.
    falls i <= k wende das Verfahren rekursiv auf Gruppe 1 an;
    falls i > k+1: verwende Verfahren rekursiv zur Bestimmung des i - (k+1)-te Element in Gruppe 2



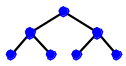
Zusammenfassung

- Sequentielle Suche
  - Default-Ansatz zur Suche
  - lineare Kosten O (n)
- Binärsuche
  - setzt Sortierung voraus
  - Divide-and-Conquer-Strategie
  - wesentlich schneller als sequentielle Suche
  - Komplexität: O (log n)
- Weitere Suchverfahren auf sortierten Arrays für Sonderfälle
  - Fibonacci-Suche
  - Sprungsuche
  - exponentielle Suche
  - Interpolationssuche
- Auswahlproblem auf unsortierter Eingabe mit linearen Kosten lösbar



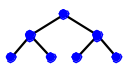
3. Verkettete Listen, Stacks, Queues

- Verkettete lineare Listen
  - Einfache Verkettung
  - Doppelt verkettete Listen
  - Vergleich der Implementierungen
  - Iterator-Konzept
- Fortgeschrittenere Kettenstrukturen
  - Selbstorganisierende (adaptive) Listen
  - Skip-Listen
- Spezielle Listen: Stack, Queue, Priority Queue
  - Operationen
  - formale ADT-Spezifikation
  - Anwendung



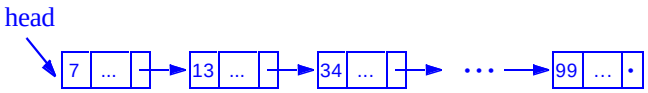
Verkettete Speicherung linearer Listen

- Sequentielle Speicherung erlaubt schnelle Suchverfahren
  - falls Sortierung vorliegt
  - da jedes Element über Indexposition direkt ansprechbar
- Nachteile der sequentiellen Speicherung
  - hoher Änderungsaufwand durch Verschiebekosten: O(n)
  - schlechte Speicherplatzausnutzung
  - inflexibel bei starkem dynamischem Wachstum
- Abhilfe: verkettete lineare Liste (Kette)
- Spezielle Kennzeichnung erforderlich für
  - Listenanfang (Anker)
  - Listenende
  - leere Liste



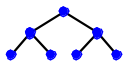
Verkettete Liste: Implementierung 1

- Implementierung 1:
  - Listenanfang wird durch speziellen Zeiger *head* (Kopf, Anker) markiert
  - Leere Liste: head = null
  - Listenende: Next-Zeiger = null



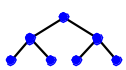
- Beispiel: Suchen von Schlüsselwert x

```
class KettenElement {
  int key;
  String wert;
  KettenElement next = null;
}
class KettenListe1 implements Liste {
  KettenElement head = null;
  ...
  public boolean search(int x) {
    KettenElement element = head;
    while ((element != null) && (element.key != x))
      element = element.next;
    return (element != null);
  }
}
```

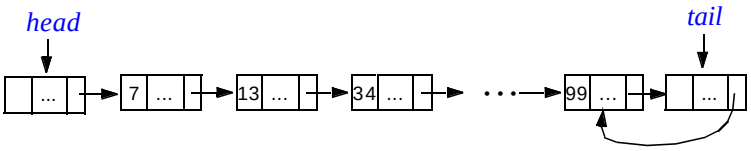
Implementierung 1(Forts.)

- nur sequentielle Suche möglich (sowohl im geordneten als auch im ungeordneten Fall) !
- Einfügen und Löschen eines Elementes mit Schlüsselwert x erfordert vorherige Suche
- Bei Listenoperationen müssen Sonderfälle stets abgeprüft werden (Zeiger auf Null prüfen etc.)
- Löschen eines Elementes an Position (Zeiger) p ?
- Hintereinanderfügen von 2 Listen ?



Verkette Liste: Implementierung 2

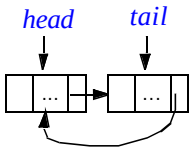
- Implementierung 2:
  - Dummy-Element am Listenanfang sowie am Listenende (Zeiger *head* und *tail*)

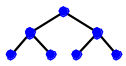


  - Next-Zeiger des Dummy-Elementes am Listenende verweist auf vorangehendes Element (erleichtert Hintereinanderfügen zweier Listen)

```
class KettenListe2 implements Liste {
  KettenElement head = null;
  KettenElement tail = null;
  /** Konstruktor */
  public KettenListe2() {
    head = new KettenElement();
    tail = new KettenElement();
    head.next = tail;
    tail.next = head;
  }...}
```

  - Leere Liste:





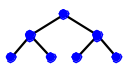
Implementierung 2 (Forts.)

- Suche von Schlüsselwert x (mit Stopper-Technik)

```
public boolean search(int x) {
  KettenElement element = head.next;
  tail.key = x;
  while (element.key != x)
    element = element.next;
  return (element != tail);
}
```

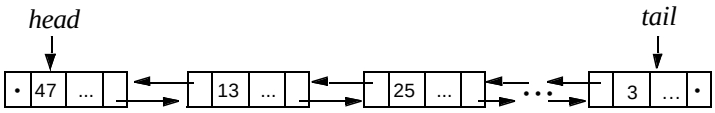
- Verketten von zwei Listen
  - Aneinanderfügen von aktueller Liste und L ergibt neue Liste

```
public KettenListe2 concat (KettenListe2 L) {
  KettenListe2 liste = new KettenListe2();
  liste.head = head;
  tail.next.next = L.head.next;
  liste.tail = L.tail;
  if (L.tail.next == L.head) // leere Liste L
    liste.tail.next = tail.next;
  return liste;
}
```

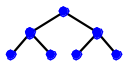



Verkettete Liste: Implementierung 3

- Implementierung 3: doppelt gekettete Liste

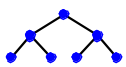



```
class KettenElement2 {
  int key;
  String wert;
  KettenElement2 next = null;
  KettenElement2 prev = null;
}

// Liste
class KettenListe3 implements Liste {
  KettenElement2 head = null;
  KettenElement2 tail = null;
  /** Konstruktor */
  public KettenListe3() {
    head = new KettenElement2();
    tail = new KettenElement2();
    head.next = tail;
    tail.prev = head;
  }
  ...
}
```

Implementierung 3 (Forts.)

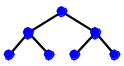
- Bewertung:
  - höherer Speicherplatzbedarf als bei einfacher Verkettung
  - Aktualisierungsoperationen etwas aufwendiger (Anpassung der Verkettung)
  - Suchaufwand in etwa gleich hoch, jedoch ggf. geringerer Suchaufwand zur Bestimmung des Vorgängers (Operation PREVIOUS (L, p))
  - geringerer Aufwand für Operation DELETE (L, p)
- Flexibilität der Doppelverkettung besonders vorteilhaft, wenn Element gleichzeitig Mitglied mehrerer Listen sein kann (Multilist-Strukturen)



Verkettete Listen

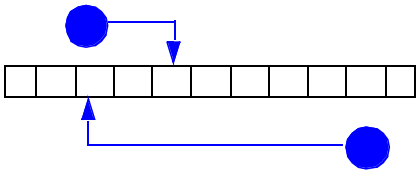
- Suchaufwand bei ungeordneter Liste
  - erfolgreiche Suche: $C_{avg} = \frac{n+1}{2}$ (Standardannahmen: zufällige Schlüsselauswahl; stochastische Unabhängigkeit der gespeicherten Schlüsselmenge)
  - erfolglose Suche: vollständiges Durchsuchen aller n Elemente
- Einfügen oder Löschen eines Elements
  - konstante Kosten für Einfügen am Listenanfang
  - Löschen verlangt meist vorherige Suche
  - konstante Löschkosten bei positionsbezogenem Löschen und Doppelverkettung
- Sortierung bringt kaum Vorteile
  - erfolglose Suche verlangt im Mittel nur noch Inspektion von (n+1)/2 Elementen
  - lineare Kosten für Einfügen in Sortierreihenfolge

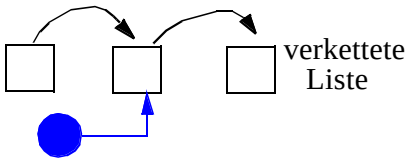
VERGLEICH der 3 Implementierungen	Implem. 1	Implem. 2	Implem. 3 (Doppelkette)
Einfügen am Listenanfang			
Einfügen an gegebener Position			
Löschen an gegebener Position			
Suchen eines Wertes			
Hintereinanderfügen von 2 Listen			



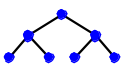
Iterator-Konzept

- Problem: Navigation der Listen ist implementationsabhängig
- Iterator-Konzept ermöglicht einheitliches sequentielles Navigieren
  - Iterator ist Objekt zum Iterieren über Kollektionen (Listen, Mengen ...)
  - mehrere Iteratoren auf einer Kollektion möglich





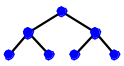
- Java-Schnittstelle für Iteratoren `java.util.Iterator` mit folgenden Methoden:
  - `boolean hasNext()`
    liefert true wenn weitere Elemente in Kollektion verfügbar, ansonsten false
  - `Object next()`
    liefert das aktuelle Element und positioniert auf das nächste Element
  - `void remove()`
    löscht das aktuelle Element



Iterator-Konzept (2)

- Implementierung am Beispiel der einfach verketteten Liste

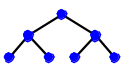
```
class KettenListe1 implements Liste {
  class ListIterator implements java.util.Iterator {
    private KettenElement element = null;
    /** Konstruktor */
    public ListIterator(KettenElement e) { element = e; }

    public boolean hasNext() { return element != null; }

    public void remove() {
      throw new UnsupportedOperationException(); }

    public Object next() {
      if(!hasNext())throw new java.util.NoSuchElementException();
      Object o = element.wert;
      element = element.next;
      return o; }
  }
  public java.util.Iterator iterator() {
    return new ListIterator(head); }
  ...
}
```

Iterator-Konzept (3)

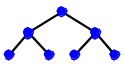
- Verwendung von Iteratoren

```
KettenListe1 liste = new KettenListe1();
...

java.util.Iterator iter = liste.iterator();
String wert = null;

while (iter.hasNext()) {
  wert = (String) iter.next();
  ...
}
```

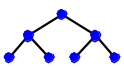
Häufigkeitsgeordnete lineare Listen

- sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind
- mittlere Suchkosten: $C_{avg}(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + n \cdot p_n$
  für Zugriffswahrscheinlichkeiten p_i
- Zur Minimierung der Suchkosten sollte Liste direkt so aufgebaut oder umorganisiert werden, daß $p_1 \geq p_2 \geq \ldots \geq p_n$
- Beispiel: Zugriffsverteilung nach 80-20-Regel
  - 80% der Suchanfragen betreffen 20% des Datenbestandes und von diesen 80% wiederum 80% (also insgesamt 64%) der Suchanfragen richten sich an 20% von 20% (insgesamt 4%) der Daten.
  - Erwarteter Suchaufwand $C_{avg}(n) =$
- Da Zugriffshäufigkeiten meist vorab nicht bekannt sind, werden selbstorganisierende (adaptive) Listen benötigt



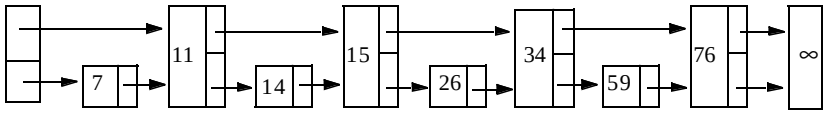
Selbstorganisierende Listen

- Ansatz 1: FC-Regel (Frequency count)
  - Führen von Häufigkeitszählern pro Element
  - Jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1
  - falls erforderlich, wird danach die Liste lokal neu geordnet, so daß die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
  - hoher Wartungsaufwand und Speicherplatzbedarf
- Ansatz 2: T-Regel (Transpose)
  - das Zielelement eines Suchvorgangs wird mit dem unmittelbar vorangehenden Element vertauscht
  - häufig referenzierte Elemente wandern (langsam) an den Listenanfang
- Ansatz 3: MF-Regel (Move-to-Front)
  - Zielelement eines Suchvorgangs wird nach jedem Zugriff an die erste Position der Liste gesetzt
  - relative Reihenfolge der übrigen Elemente bleibt gleich
  - in jüngster Vergangenheit referenzierte Elemente sind am Anfang der Liste (Lokalität kann gut genutzt werden)

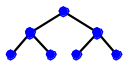


Skip-Listen

- Ziel: verkettete Liste mit logarithmischem Aufwand für Suche, Einfügen und Löschen von Schlüsseln (Wörterbuchproblem)
  - Verwendung sortierter verkettet gespeicheter Liste mit zusätzlichen Zeigern
- Prinzip
  - Elemente werden in Sortierordnung ihrer Schlüssel verkettet
  - Führen *mehrerer* Verkettungen auf unterschiedlichen Ebenen:
    Verkettung auf Ebene 0 verbindet alle Elemente;
    Verkettung auf Ebene 1 verbindet jedes zweite Element; •••
    Verkettung auf Ebene i verbindet jedes 2^i-te Element

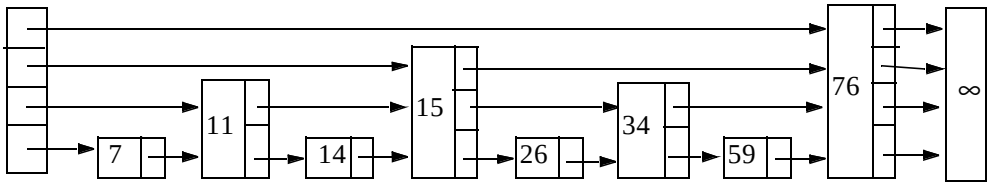


- Suche:
  - beginnt auf oberster Ebene bis Element E gefunden wird, dessen Schlüssel den Suchschlüssel übersteigt (dabei werden viele Elemente übersprungen)
  - Fortsetzung der Suche auf darunterliegender Ebene bei Elementen, die nach dem Vorgänger von E folgen
  - Fortsetzung des Prozesses bis auf Ebene 0

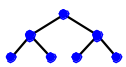


Skip-Listen (2)

- Perfekte Skip-Liste:
  - Anzahl der Ebenen (Listenhöhe): 1 + log n



  - max. Gesamtanzahl der Zeiger:
  - Suche: O (log n)
- Perfekte Skip-Listen zu aufwendig bezüglich Einfügungen und Löschvorgängen
  - vollständige Reorganistion erforderlich
  - Kosten O (n)

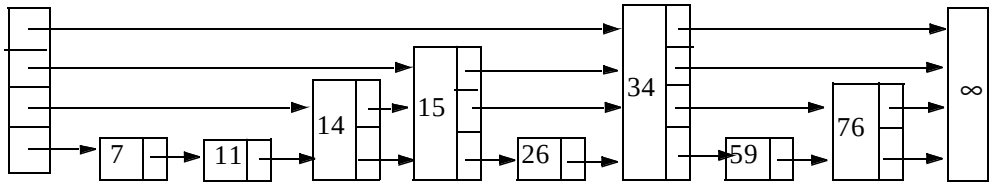


Skip-Listen (3)

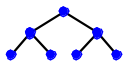
- Abhilfe: Randomisierte Skip-Listen
  - strikte Zuordnung eines Elementes zu einer Ebene ("Höhe") wird aufgegeben
  - Höhe eines neuen Elementes x wird nach Zufallsprinzip ermittelt, jedoch so daß die relative Häufigkeit der Elemente pro Ebene (Höhe) eingehalten wird, d.h.

    $$P(\text{Höhe von } x = i) = 1 / 2^i \text{ (für alle } i)$$

  - somit entsteht eine "zufällige" Struktur der Liste



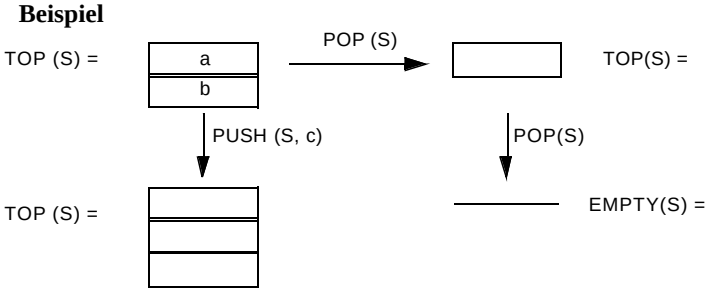
- Kosten für Einfügen und Löschen im wesentlichen durch Aufsuchen der Einfügeposition bzw. des Elementes bestimmt: O(log N)

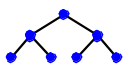


Stacks

- Synonyme: Stapel, Keller, LIFO-Liste usw.
- Stack kann als spezielle Liste aufgefaßt werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt, vorgenommen werden
- Stack-Operationen (ADT):
  - CREATE: Erzeugt den leeren Stack
  - INIT(S): Initialisiert S als leeren Stack
  - PUSH(S, x): Fügt das Element x als oberstes Element von S ein
  - POP(S): Löschen des Elementes, das als letztes in den Stack S eingefügt wurde
  - TOP(S): Abfragen des Elementes, das als letztes in den Stack S eingefügt wurde
  - EMPTY(S): Abfragen, ob der Stack S leer ist
- alle Operationen mit konstanten Kosten realisierbar: O(1)

Beispiel





Stacks (2)

- formale ADT-Spezifikation zur Festlegung der implementierungsunabhängigen Stack-Eigenschaften
  - ELEM = Wertebereich der Stack-Elemente
  - STACK = Menge der Zustände, in denen sich der Stack befinden kann
  - leerer Stack: $s_0 \in$ STACK
  - Stack-Operationen werden durch ihre Funktionalität charakterisiert. Ihre Semantik wird durch Axiome festgelegt.
- Definitionen:

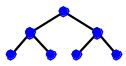
Datentyp STACK
Basistyp ELEM

Operationen:

CREATE:		→	STACK;
INIT:	STACK	→	STACK;
PUSH:	STACK × ELEM	→	STACK;
POP:	STACK - {s_0}	→	STACK;
TOP:	STACK - {s_0}	→	ELEM;
EMPTY:	STACK	→	{TRUE, FALSE}.

Axiome:

CREATE = s_0;
EMPTY (CREATE) = TRUE;
∀s ∈ STACK, ∀x ∈ ELEM:
INIT (s) = s_0;
EMPTY (PUSH(s , x)) = FALSE;
TOP (PUSH (s , x)) = x;
POP (PUSH (s , x)) = s;
NOT EMPTY (s) ⇒ PUSH (POP(s) , TOP(s)) = s



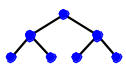
Stacks (3)

- Interface-Definition

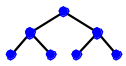
```
public interface Stack {
  public void push(Object o) throws StackException;
  public Object pop() throws StackException;
  public Object top() thrwos StachException;
  public boolean isempty();
}
```

- Array-Implementierung des Stack-Interface

```
public class ArrayStack implements Stack {
  private Object elements[] = null;
  private int count = 0;
  private final int defaultSize = 100;

  public ArrayStack(int size) {
    elements = new Object[size];
  }

  public ArrayStack() {
    elements = new Object[defaultSize];
  }
```

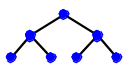



```
public void push(Object o) throws StackException {
  if(count == elements.length-1)
    throw new StackException("Stack voll!");
  elements[count++] = o;
}

public Object pop() throws StackException {
  if(isempty())
    throw new StackException("Stack leer!");
  Object o = elements[--count];
  elements[count] = null; // Freigeben des Objektes
  return o;
}

public Object top() throws StackException {
  if(isempty())
    throw new StackException("Stack leer!");
  return elements[count-1];
}

public boolean isempty() {
  return count == 0;
}
```

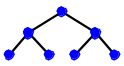
Stacks (4)

- Anwendungsbeispiel 1: Erkennen wohlgeformter Klammerausdrücke
- Definition
  - () ist ein wohlgeformter Klammerausdruck (wgK)
  - Sind w1 und w2 wgK, so ist auch ihre Konkatenation w1 w2 ein wgK
  - Mit w ist auch (w) ein wgK
  - Nur die nach den vorstehenden Regeln gebildeten Zeichenreihen bilden wgK
- Lösungsansatz
  - Speichern der öffnenden Klammern in Stack
  - Entfernen des obersten Stack-Elementes bei jeder schließenden Klammer
  - wgK liegt vor, wenn Stack am Ende leer ist



- Realisierung

```
public boolean wgK(String ausdruck) {
  Stack stack = new ArrayStack();
  char ch;
  for (int pos=0; pos < ausdruck.length(); pos++) {
    ch = ausdruck.charAt(pos);
    if (ch == '(') stack.push(new Character(ch));
    else if (ch == ')') {
      if (stack.isempty()) return false;
      else stack.pop();
    }
  }
  if (stack.isempty()) return true;
  else return false;
}
```

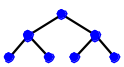



Stacks (5)

- Anwendungsbeispiel 2: Berechnung von Ausdrücken in Umgekehrter Polnischer Notation (Postfix-Ausdrücke)
- Beispiel: (a+b) x (c+d/e) => a b + c d e / + x
- Lösungsansatz
  - Lesen des Ausdrucks von links nach rechts
  - Ist das gelesene Objekt ein Operand, wird es auf den STACK gebracht
  - Ist das gelesene Objekt ein m-stelliger Operator, dann wird er auf die m obersten Elemente des Stacks angewandt. Das Ergebnis ersetzt diese m Elemente
- Abarbeitung des Beispielausdrucks:

UPN	a	b	+	c	d	e	/	+	x

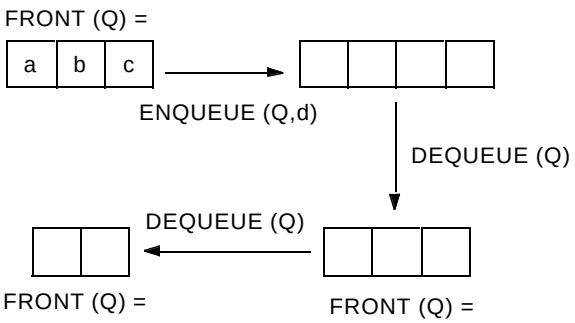
Platz 1									
Platz 2									
Platz 3									
Platz 4									



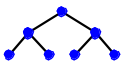
Schlangen

- Synonyme: FIFO-Schlange, Warteschlange, Queue
- spezielle Liste, bei der die Elemente an einem Ende (hinten) eingefügt und am anderen Ende (vorne) entfernt werden

Beispiel:



- Operationen:
  - CREATE: Erzeugt die leere Schlange
  - INIT (Q): Initialisiert Q als leere Schlange
  - ENQUEUE (Q, x): Fügt das Element x am Ende der Schlange Q ein
  - DEQUEUE (Q): Löschen des Elementes, das am längsten in der Schlange verweilt (erstes Element)
  - FRONT (Q): Abfragen des ersten Elementes in der Schlange
  - EMPTY (Q): Abfragen, ob die Schlange leer ist



Schlangen (2)

- formale ADT-Spezifikation
  - ELEM = Wertebereich der Schlangen-Elemente
  - QUEUE = Menge der möglichen Schlangen-Zustände
  - leere Schlange: $q_0 \in$ QUEUE

Datentyp QUEUE
Basistyp ELEM

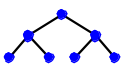
Operationen:

CREATE	:	$\rightarrow$	QUEUE;
INIT	:	QUEUE$\rightarrow$	QUEUE;
ENQUEUE	:	QUEUE $\times$ ELEM$\rightarrow$	QUEUE;
DEQUEUE	:	QUEUE - $\{q_0\}\rightarrow$	QUEUE;
FRONT	:	QUEUE - $\{q_0\}\rightarrow$	ELEM;
EMPTY	:	QUEUE$\rightarrow$	{TRUE, FALSE}.

Axiome:

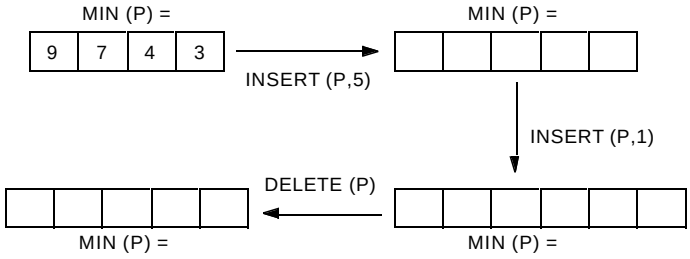
CREATE= q_0;
EMPTY (CREATE) = TRUE;

$\forall q \in$ QUEUE, $\forall x \in$ ELEM:
INIT (q) = q_0;
EMPTY (ENQUEUE(q, x)) = FALSE;
EMPTY (q) $\Rightarrow$ FRONT (ENQUEUE(q , x)) = x;
EMPTY (q) $\Rightarrow$ DEQUEUE (ENQUEUE(q , x) = q;
NOT EMPTY (q) $\Rightarrow$ FRONT (ENQUEUE(q , x)) = FRONT(q);
NOT EMPTY (q) $\Rightarrow$ DEQUEUE (ENQUEUE(q , x)) = ENQUEUE(DEQUEUE(q), x).

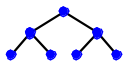


Vorrangwarteschlangen

- Vorrangwarteschlange (priority queue)
  - jedes Element erhält Priorität
  - entfernt wird stets Element mit der höchsten Priorität (Aufgabe des FIFO-Verhaltens einfacher Warteschlangen)



- Operationen:
  - CREATE: Erzeugt die leere Schlange
  - INIT(P): Initialisiert P als leere Schlange
  - INSERT(P, x): Fügt neues Element x in Schlange P ein
  - DELETE(P): Löschen des Elementes mit der höchsten Priorität aus P
  - MIN(P): Abfragen des Elementes mit der höchsten Priorität
  - EMPTY(P): Abfragen, ob Schlange P leer ist.
- Sortierung nach Prioritäten beschleunigt Operationen DELETE und MIN auf Kosten von INSERT



Vorrangwarteschlangen (2)

- formale ADT-Spezifikation :

Datentyp PQUEUE
Basistyp ELEM
(besitzt totale Ordnung ≤)

leere Vorrangwarteschlange:
$p_0 \in$ PQUEUE

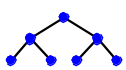
Operationen:

CREATE:		→	PQUEUE;
INIT:	PQUEUE	→	PQUEUE;
INSERT:	PQUEUE × ELEM	→	PQUEUE;
DELETE:	PQUEUE - $\{p_0\}$	→	PQUEUE;
MIN:	PQUEUE - $\{p_0\}$	→	ELEM;
EMPTY:	PQUEUE	→	{TRUE, FALSE}.

Axiome:

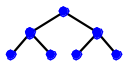
CREATE = p_0;
EMPTY (CREATE) = TRUE;

$\forall p \in$ PQUEUE, $\forall x \in$ ELEM:
INIT (p) = p_0;
EMPTY (INSERT (p, x)) = FALSE;
EMPTY (p) ⇒ MIN (INSERT (p, x)) = x;
EMPTY (p) ⇒ DELETE (INSERT (p, x)) = p;
NOT EMPTY (p) ⇒ IF x ≤ MIN (p) THEN MIN (INSERT (p, x)) = x
ELSE MIN (INSERT (p, x)) = MIN(p);
NOT EMPTY (p) ⇒ IF x ≤ MIN (p) THEN DELETE (INSERT (p , x)) = p
ELSE DELETE (INSERT (p, x)) = INSERT (DELETE (p), x);



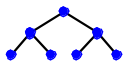
Zusammenfassung

- Verkettete Listen
  - dynamische Datenstrukturen mit geringem Speicheraufwand und geringem Änderungsaufwand
  - Implementierungen: einfach vs. doppelt verkettete Listen
  - hohe Flexibilität
  - hohe Suchkosten
- Iterator-Konzept: implementierungsunabhängige Navigation in Kollektionen (u.a. Listen)
- Adaptive (selbstorganisierende) Listen erlauben reduzierte Suchkosten
  - Nutzung von Lokalität bzw. ungleichmäßigen Zugriffshäufigkeiten
  - Umsetzung z.B. über Move-to-Front oder Transpose
- Skip-Listen
  - logarithmische Suchkosten
  - randomisierte statt perfekter Skip-Listen zur Begrenzung des Änderungsaufwandes
- ADT-Beispiele: Stack, Queue, Priority Queue
  - spezielle Listen mit eingeschränkten Operationen (LIFO bzw. FIFO)
  - formale ADT-Spezifikation zur Festlegung der implementierungsunabhängigen Eigenschaften
  - effiziente Implementierbarkeit der Operationen: O(1)
  - zahlreiche Anwendungsmöglichkeiten



4. Sortierverfahren

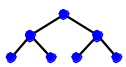
- Einführung
- Elementare Sortierverfahren
  - Sortieren durch direktes Auswählen (Straight Selection Sort)
  - Sortieren durch Vertauschen (Bubble Sort)
  - Sortieren durch Einfügen (Insertion Sort)
- Shell-Sort (Sortieren mit abnehmenden Inkrementen)
- Quick-Sort
- Auswahl-Sortierung (Turnier-Sortierung, Heap-Sort)
- Sortieren durch Streuen und Sammeln
- Merge-Sort
- Externe Sortierverfahren
  - Ausgeglichenes k-Wege-Merge-Sort
  - Auswahl-Sortierung mit Ersetzung (Replacement Selection)



Anwendungen

- Sortierung ist fundamentale Operation in zahllosen System- und Anwendungsprogrammen: dominierender Anteil in Programmlaufzeiten
- Beispiele
  - Wörter in Lexikon
  - Bibliothekskatalog
  - Kontoauszug (geordnet nach Datum der Kontobewegung)
  - Sortierung von Studenten nach Namen, Notendurchschnitt, Semester, ...
  - Sortierung von Adressen / Briefen nach Postleitzahlen, Ort, Straße ...
  - indirekte Nutzung u.a. bei Duplikaterkennung (z.B. # unterschiedlicher Wörter in einem Text)
- Naive Implementierung zur Duplikaterkennung

```
public static boolean duplicates( Object [ ] A ) {
  for( int i = 0; i < A.length; i++ )
    for( int j = i + 1; j < A.length; j++ )
      if( A[i].equals( A[j] ) )
        return true;   // Duplicate found
  return false;        // No duplicates found
}
```

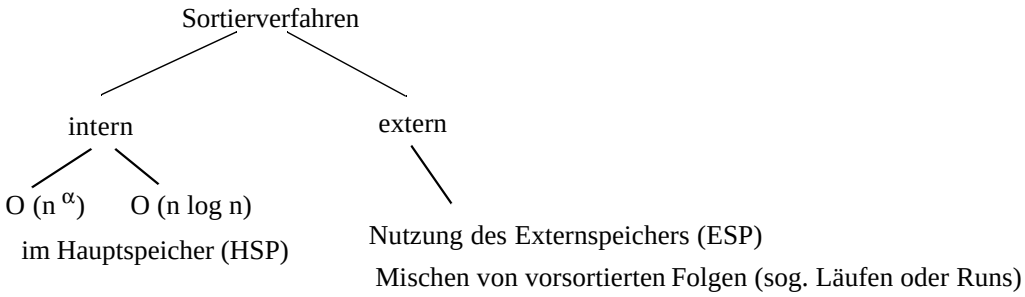



Sortierverfahren

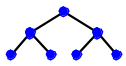
- allgemeine Problemstellung
  - Gegeben: Folge von Sätzen S_1, ... S_n, wobei jeder Satz S_i Schlüssel K_i besitzt
  - Gesucht: Permutation π der Zahlen von 1 bis n, welche aufsteigende Schlüsselreihenfolge ergibt:

$$K_{\pi(1)} \leq K_{\pi(2)} \leq \ldots \leq K_{\pi(n)}$$

- Interne vs. externe Sortierverfahren



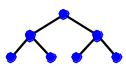
- einfache vs. spezielle Sortierverfahren
  - Annahmen über Datenstrukturen (z.B. Array) oder Schlüsseleigenschaften



Sortierverfahren (2)

- Wünschenswert: *stabile* Sortierverfahren, bei denen die relative Reihenfolge gleicher Schlüsselwerte bei der Sortierung gewahrt bleibt
- Speicherplatzbedarf am geringsten für Sortieren am Ort (“in situ”)
- Weitere Kostenmaße
  - #Schlüsselvergleiche C (C_{min}, C_{max}, C_{avg})
  - #Satzbewegungen M (M_{min}, M_{max}, M_{avg})
- Satz:

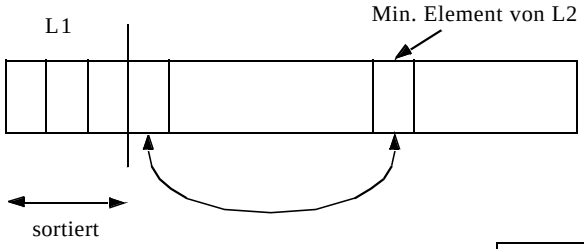
  Jedes allgemeine Sortierverfahren, welches zur Sortierung nur Vergleichsoperationen zwischen Schlüsseln (sowie Tauschoperationen) verwendet, benötigt sowohl im mittleren als auch im schlechtesten Fall wenigstens Ω (n • log n) Schlüsselvergleiche



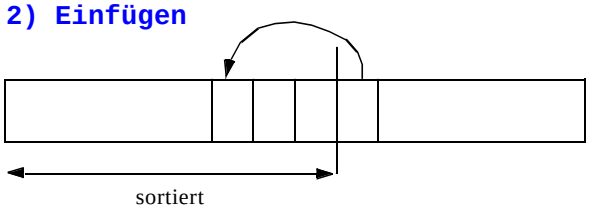
Klassifizierung von Sortiertechniken

Sortieren durch . . .

1) Auswählen

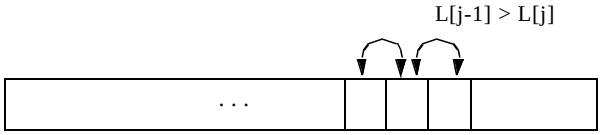


2) Einfügen

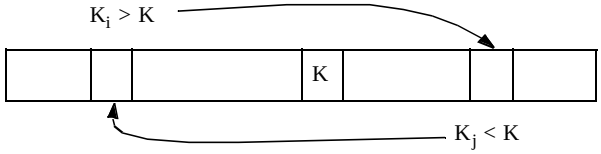


3) Austauschen

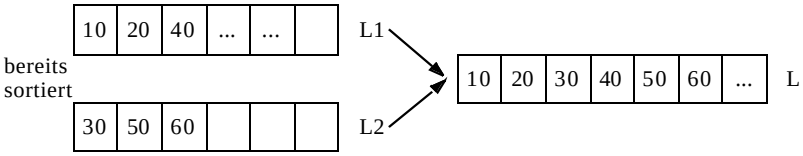
a) lokal

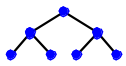


b) entfernt



4) Mischen

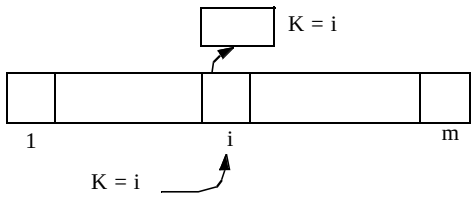




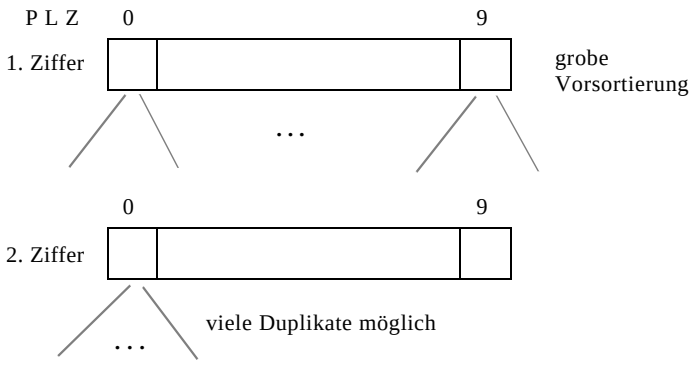
Klassifizierung von Sortiertechniken (2)

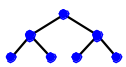
5. Streuen und Sammeln

- begrenzter Schlüsselbereich m, z. B. 1 - 1000
- relativ dichte Schlüsselbelegung n ≤ m
- Duplikate möglich (n > m)
- lineare Sortierkosten !



6. Fachverteilen (z. B. Poststelle)

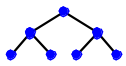




Basisklasse für Sortieralgorithmen

```
public abstract class SortAlgorithm {

  static void swap (Object A[], int i, int j) {
    Object o = A[i];
    A[i] = A[j];
    A[j] = o;
  }

  /** Sortiert das übergebene Array;
   *  Element an Position 0 bleibt unberücksichtigt (Stopper...)!
   */
  public static void sort (Orderable A[]) {}

  public static void print(Orderable A[]) {
    for (int i=1; i<A.length; i++)
      System.out.println(A[i].getKey());
  }
}
```

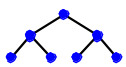
- auf zu sortierenden Objekten muß Ordnung definiert sein



Sortieren durch Auswählen (Selection Sort)

- Idee
  - Auswahl des kleinsten Elementes im unsortierten Teil der Liste
  - Austausch mit dem ersten Element der unsortierten Teilliste

27	75	99	3	45	12	87



Sortieren durch Auswählen (2)

- Sortierprozedur

```
public class SelectionSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i = 1; i < A.length-1; i++) {
      int min = i;       // Suche kleinstes Element
      for (int j = i+1; j < A.length; j++) {
        if (A[j].less(A[min])) {
          min = j;
        }
      }
      swap (A, i, min);  // tausche aktuelles mit kleinstem Element
    }
  }
}
```

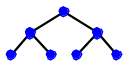
- Anzahl Schlüsselvergleiche

$C_{min}(n) = C_{max}(n) =$

- Anzahl Satzbewegungen (durch Swap):

$M_{min}(n) = M_{max}(n) =$

- in-situ-Verfahren, nicht stabil

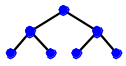


Sortieren durch Vertauschen (Bubble Sort)

- Idee
  - Vertauschen benachbarter Elemente, die nicht in Sortierordnung
  - pro Durchgang wandert größtes Element der noch unsortierten Teilliste nach “oben”
  - Sortierung endet, wenn in einem Durchgang keine Vertauschung mehr erfolgte

1						N
27	75	99	3	45	12	87

- Variation: Shaker-Sort (Durchlaufrichtung wechselt bei jedem Durchgang)



Bubble Sort (2)

```
public class BubbleSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i=A.length-1; i>1; i--) {
      for (int j=1; j<i; j++) {
        if (A[j].greater(A[j+1]))
          swap (A, j, j+1);  // Vertauschen benachbarter Elemente
      }
    }
  }
}
```

- Kosten

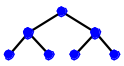
  $C_{min}(n) = C_{max}(n) = C_{avg}(n) =$

  $M_{min}(n) =$

  $M_{max}(n) =$

  $M_{avg}(n) =$

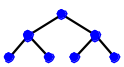
- in-situ-Verfahren, stabil
- Vorsortierung kann genutzt werden



Sortieren durch Einfügen (Insertion Sort)

- Idee
  - i-tes Element der Liste x (1. Element der unsortierten Teilliste) wird an der richtigen Stelle der bereits sortierten Teilliste (1 bis i-1) eingefügt
  - Elemente in sortierter Teilliste mit höherem Schlüsselwert als x werden verschoben

1						N
27	75	99	3	45	12	87



Insertion Sort (2)

```
public class InsertionSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i = 2; i < A.length; i++) {
      A[0] = A[i];  // Speichere aktuelles Element an reservierter Pos.
      int j = i - 1;
      while ((j > 0) && A[j].greater(A[0])) {
        A[j+1] = A[j];  // Verschiebe größeres Elem. eine Pos. nach rechts
        j--;
      }
      A[j+1] = A[0];   // Füge zwischengespeichertes Elem. ein
    }
  }
}
```

- Kosten

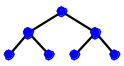
  $C_{min}(n) =$

  $C_{max}(n) =$

  $M_{min}(n) =$

  $M_{max}(n) =$

- in-situ-Verfahren, stabil



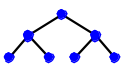
Vergleich der einfachen Sortierverfahren

- #Schlüsselvergleiche

	C_{min}	C_{avg}	C_{max}
Direktes Auswählen	n (n - 1) / 2	n (n - 1) / 2	n (n - 1) / 2
Bubble Sort	n (n - 1) / 2	n (n - 1) / 2	n (n - 1) / 2
Einfügen	n - 1	$(n^2 + 3n + 4) / 4$	$(n^2 + n - 2) / 2$

- #Satzbewegungen

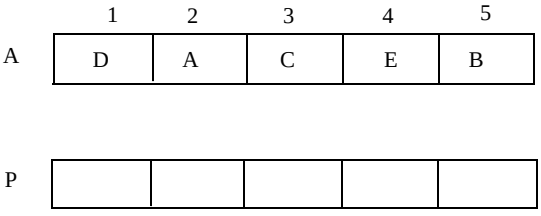
	M_{min}	M_{avg}	M_{max}
Direktes Auswählen	3 (n - 1)	3 (n - 1)	3 (n - 1)
Bubble Sort	0	3n (n - 1) / 4	3n (n - 1) / 2
Einfügen	3 (n - 1)	$(n^2 + 11 n + 12) / 4$	$(n^2 + 5n - 6) / 2$



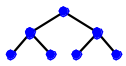
Sortieren von großen Datensätzen (Indirektes Sortieren)

- Indirektes Sortieren erlaubt, Kopieraufwand für *jedes* Sortierverfahrens auf lineare Kosten O (n) zu beschränken
- Führen eines Hilfsfeldes von Indizes auf das eigentliche Listenfeld

Liste	A [1..n]
Pointerfeld	P [1..n] (Initialisierung P[i] = i)
Schlüsselzugriff:	`A [i] -> A [P[i]]`
Austausch:	`swap (A, i, j) ->` `swap (P , i, j)`

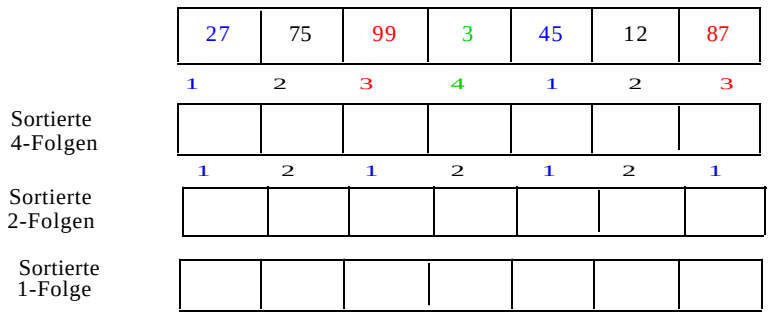


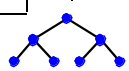
- Sortierung erfolgt lediglich auf Indexfeld
- abschließend erfolgt linearer Durchlauf zum Umkopieren der Sätze



Shell-Sort (Shell, 1957) (Sortieren mit abnehmenden Inkrementen)

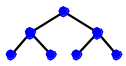
- Idee
  - Sortierung in mehreren Stufen “von grob bis fein”
  - Vorsortierung reduziert Anzahl von Tauschvorgängen
- Vorgehensweise:
  - Festlegung von t Inkrementen (Elementabständen) h_i mit $h_1 > h_2 > \ldots . h_t = 1$ zur Bildung von disjunkten Teillisten
  - Im i-ten Schritt erfolgt unabhängiges Sortieren aller h_i-Folgen (mit Insertion Sort)
  - Eine Elementbewegung bewirkt Sprung um h_i Positionen
  - Im letzten Schritt erfolgt “normales” Sortieren durch Einfügen
- Beispiel:
  $h_1 = 4; h_2 = 2; h_3 = 1$





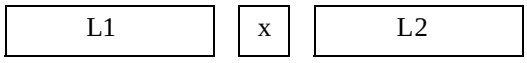
Shell-Sort (2)

- Aufwand wesentlich von Wahl der Anzahl und Art der Schrittweiten abhängig
- Insertion Sort ist Spezialfall (t=1)
- Knuth [Kn73] empfiehlt
  - Schrittweitenfolge von 1, 3, 7, 15, 31 . . . mit $h_{i-1} = 2 \cdot h_i + 1$, $h_t = 1$ und $t = \lfloor \log_2 n \rfloor - 1$
  - Aufwand O ($n^{1.2}$)
- Andere Vorschläge erreichen O ($n \log^2 n$)
  - Inkremente der Form $2^p 3^q$

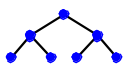


Quick-Sort

- Hoare, 1962
- andere Bezeichnung: Partition-Exchange-Sort
- Anwendung der Divide-and-Conquer-Strategie auf Sortieren durch Austauschen:
  - Bestimmung eines Pivot-Elementes x in L
  - Zerlegung der Liste in zwei Teillisten L1 und L2 durch Austauschen, so daß linke (rechte) Teilliste L1 (L2) nur Schlüssel enthält, die kleiner (größer oder gleich) als x sind

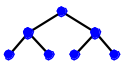


  - Rekursive Sortierung der Teillisten, bis nur noch Listen der Länge 1 verbleiben
- Realisierung der Zerlegung
  - Duchlauf des Listenbereiches von links über Indexvariable i, solange bis L [i] >= x vorliegt
  - Duchlauf des Listenbereiches von rechts über Indexvariable j, solange bis L [j] <= x vorliegt
  - Austausch von L[i] und L [j]
  - Fortsetzung der Durchläufe bis i > j gilt



Quick-Sort-Beispiel

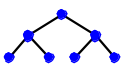
67	58	23	44	91	11	30	54



Quick-Sort (2)

```
public class QuickSort extends SortAlgorithm {

  public static void sort (Orderable A[], int l, int r) {
    int i=l;
    int j=r;
    if (r <= l) return;  // Listenlänge < 2 -> Ende
    A[0] = A[(l+r)/2];   // speichere Pivot-Element in Position 0
    do {
      while (A[i].less(A[0])) i++;
      while (A[j].greater(A[0])) j--;
      if (i <= j) {
        swap (A, i, j);
        i++; j--;
      }
    } while (i <= j);
    sort (A, l, j);
    sort (A, i, r);
  }

  public static void sort (Orderable A[]) {
    sort(A, 1, A.length-1);
  } }
```

- In-situ-Verfahren; nicht “stabil”



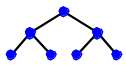
Quick-Sort (3)

- Kosten am geringsten, wenn Teillisten stets gleichlang sind, d.h. wenn das Pivot-Element dem mittleren Schlüsselwert (Median) entspricht
  - Halbierung der Teillisten bei jeder Zerlegung
  - Kosten O (n log n)
- Worst-Case
  - Liste der Länge k wird in Teillisten der Längen 1 und k-1 zerlegt (z.B. bei bereits sortierter Eingabe und Wahl des ersten oder letzten Elementes als Pivot-Element)
  - Kosten O (n^2)
- Wahl des Pivot-Elementes von entscheidender Bedeutung

Sinnvolle Methoden
  - mittleres Element
  - Mittlerer Wert von k Elementen (z.B. k=3)

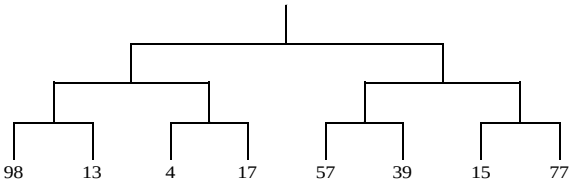
  => bei fast allen Eingabefolgen können Kosten in der Größenordnung O (n log n) erzielt werden

- Zahlreiche Variationen von Quick-Sort
  (z.B. Behandlung von Duplikaten, Umschalten auf elementares Sortierverfahren für kleine Teillisten)

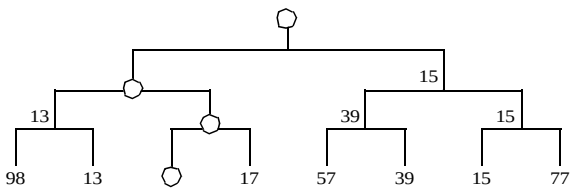


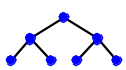
Turnier-Sortierung

- Maximum- bzw. Minimum-Bestimmung einer Sortierung analog zur Siegerermittlung bei Sportturnieren mit KO-Prinzip
  - paarweise Wettkämpfe zwischen Spielern/Mannschaften
  - nur Sieger kommt weiter
  - Sieger des Finales ist Gesamtsieger
- Zugehörige Auswahlstruktur ist ein binärer Baum
- Aber: der zweite Sieger (das zweite Element der Sortierung) ist nicht automatisch der Verlierer im Finale



- Stattdessen: Neuaustragung des Wettkampfes auf dem Pfad des Siegers (ohne seine Beteiligung)
  - Pfad für Wurzelelement hinabsteigen und Element jeweils entfernen
  - Neubestimmung der Sieger





Turnier-Sortierung (2)

- Algorithmus TOURNAMENT SORT:

  "Spiele ein KO-Turnier und erstelle dabei einen binären Auswahlbaum"

  FOR I := 1 **TO** n **DO**

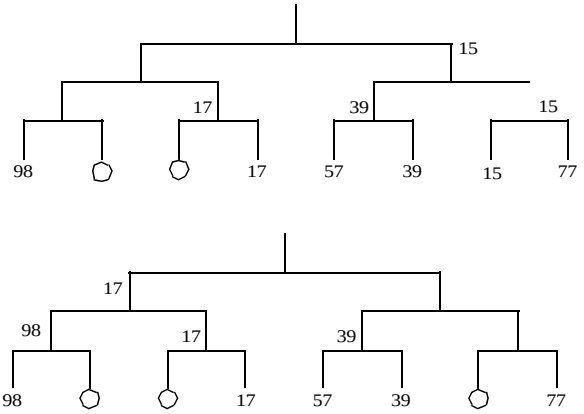
  "Gib Element an der Wurzel aus"

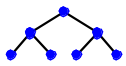
  "Steige Pfad des Elementes an der Wurzel hinab und lösche es"

  "Steige Pfad zurück an die Wurzel und spiele ihn dabei neu aus"

  END

- Weitere Schritte im Beispiel





Turnier-Sortierung (3)

- Anzahl Vergleiche (Annahme $n = 2^k$)
  - #Vergleiche für Aufbau des initialen Auswahlbaumes

$$2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

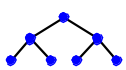
  - pro Schleifendurchlauf Absteigen und Aufsteigen im Baum über k Stufen:

$$2k = 2 \log_2 n \text{ Vergleiche}$$

  - Gesamtaufwand:

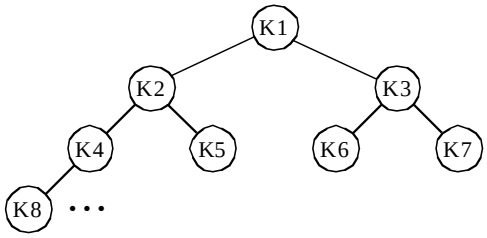
- Platzbedarf für Auswahlbaum

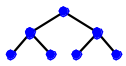
$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 = 2n - 1$$



Heap-Sort (Williams, 1964)

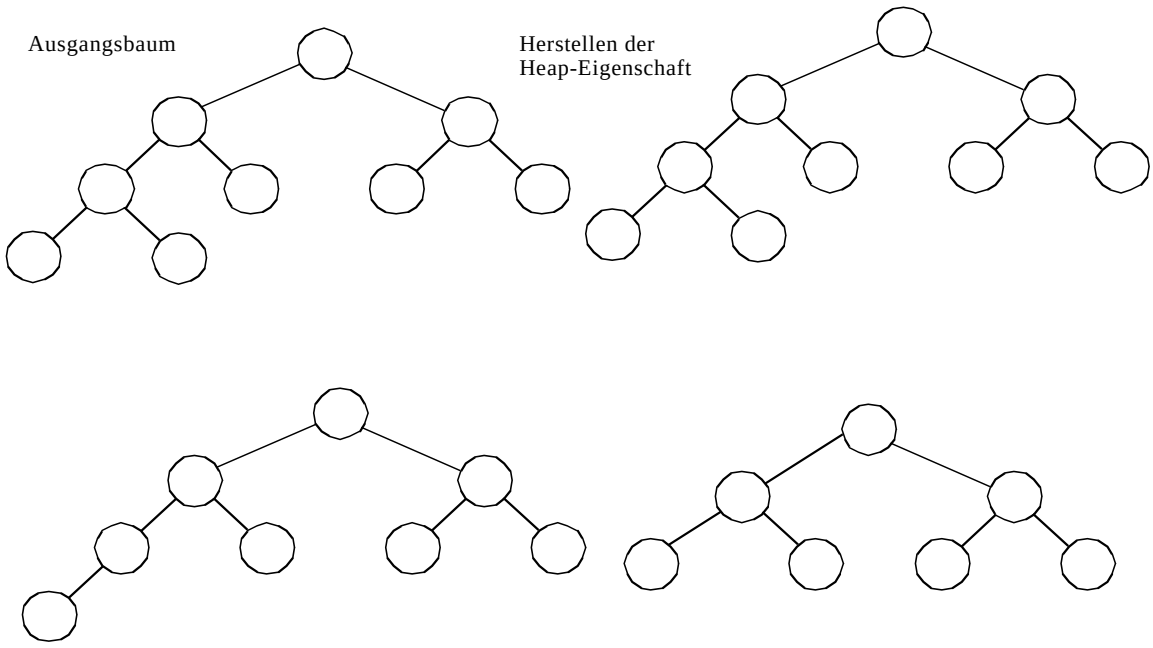
- Reduzierung des Speicherplatzbedarfs gegenüber Turnier-Sort, indem "Löcher" im Auswahlbaum umgangen werden
- *Heap (Halde):*
  - hier: binärer Auswahlbaum mit der Eigenschaft, daß sich das größte Element jedes Teilbaumes in dessen Wurzel befindet
  - => Baumwurzel enthält größtes Element
- Vorgehensweise
  - Repräsentation der Schlüsselfolge K_1, K_2, K_3, ... in einem Binärbaum minimaler Höhe nach nebenstehendem Schema
  - *Herstellen der Heap-Eigenschaft:* ausgehend von Blättern durch "Absinken" von Knoteninhalten, welche kleiner sind als für einen der direkten Nachfolgerknoten
  - Ausgabe des Wurzelelementes
  - Ersetzen des Wurzelelementes mit dem am weitesten rechts stehenden Element der untersten Baumebene
  - Wiederherstellen der Heap-Eigenschaft durch "Absenken" des Wurzelelementes; Ausgabe der neuen Wurzel usw. (solange bis Baum nur noch 1 Element enthält)

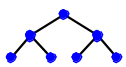




Heap-Sort (2)

Beispiel für Schlüsselfolge 57, 16, 62, 30, 80, 7, 21, 78, 41





Heap-Sort (3)

- Effiziente Realisierbarkeit mit Arrays
  - Wurzelelement = erstes Feldelement A [1]
  - direkter Vaterknoten von A [i] ist A [i / 2]
  - *Heap-Bedingung:* A [i] <= A [i/2]
- Beispiel (Ausgangsliste 57, 16, 62,30, 80, 7, 21, 78, 41)

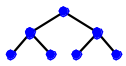
Heap-Darstellung

1	2	3	4	5	6	7	8	9
80	78	62	57	16	7	21	30	41

- Algorithmus

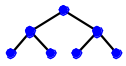
1. Aufbau des Heaps durch Absenken aller Vaterknoten A[n / 2] bis A [1]
2. Vertauschen von A [1] mit A [n] (statt Entfernen der Wurzel)
3. Reduzierung der Feldlänge n um 1
4. Falls n > 1:
   - Absenken der neuen Wurzel A [1]
   - Gehe zu 2

1	2	3	4	5	6	7	8	9



Heap-Sort (4)

```
public class HeapSort extends SortAlgorithm {
  /** Absenken des i-ten Elements im Bereich 2*i bis n */
  static void sink (Orderable A[], int i, int n) {
    while (2*i <= n) {       // i hat Sohn
      int j = 2*i;           // j zeigt auf linken Sohn
      if ((j < n) && A[j].less(A[j+1]))
        j = j + 1;           // rechter Sohn ist größer als linker
      if (A[i].less(A[j])) { // aktuelles Element kleiner als Söhne
        swap(A,i,j);         // tausche mit dem größten Sohn
        i = j;               // i zeigt wieder auf aktuelles Element
      } else i = n; }}       // Söhne kleiner als Element -> Ende

  /** Stelle Heap-Eigenschaften im Array her. */
  static void makeHeap (Orderable A[]) {
    int n = A.length-1;
    for (int i = n/2; i > 0; i--)
      sink(A,i,n); }

  public static void sort (Orderable A[]) {
    int n = A.length-1;
    makeHeap(A);                      // stelle Heap-Eigenschaft her
    for (int i = n; i > 1; i--) {     // sortiere
      swap(A,1,i);
      sink(A,1,i-1); }}}
```



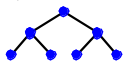
Heap-Sort (5)

- Für $2^{k-1} \le n < 2^k$ gilt
  - Baum hat die Höhe k
  - Anzahl der Knoten auf Stufe i $(0 \le i \le k-1)$ ist 2^i
- Kosten der Prozedur Sink
  - Schleife wird höchstens h - 1 mal ausgeführt (h ist Höhe des Baumes mit Wurzelelement i)
  - Zeitbedarf O(h) mit h <= k
- Kosten zur Erstellung des anfänglichen Heaps
  - Absenken nur für Knoten mit nicht-leeren Unterbäumen (Stufe k-2 bis Stufe 0)
  - Kosten für einen Knoten auf der Stufe k-2 betragen höchstens $1 \cdot c$ Einheiten, die Kosten für die Wurzel $(k-1) \cdot c$ Einheiten.
  - max. Kosten insgesamt

$$2^{k-2} \cdot 1 \cdot c + 2^{k-3} \cdot 2 \cdot c + \ldots + 2^0 \cdot (k-1) \cdot c =$$

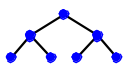
$$\sum_{i=1}^{k-1} c \cdot i \cdot 2^{k-i-1} = c \cdot 2^{k-1} \sum_{i=1}^{k-1} i \cdot 2^{-i} < c \cdot \frac{n}{2} \cdot \sum_{i=1}^{k-1} i \cdot 2^{-i}$$

  - Gesamtkosten O (n)
- Kosten der Sortierung (zweite FOR-Schleife): n-1 Aufrufe von Sink mit Kosten von höchstens O (k)=O ($\log_2$ n)
- maximale Gesamtkosten von HEAPSORT: O(n) + O(n $\log_2$n) = O(n $\log_2$n).



Sortieren durch Streuen und Sammeln (Distribution-Sort, Bucket-Sort)

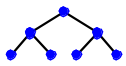
- lineare Sortierkosten O (n) !
- kein allgemeines Sortierverfahren, sondern beschränkt auf kleine und zusammenhängende Schlüsselbereiche 1..m
- Die Verteilung der Schlüsselwerte wird für alle Werte bestimmt und daraus die relative Position jedes Eingabewertes in der Sortierfolge bestimmt
- Vorgehensweise
  - Hilfsfeld COUNT [1..m]
  - Bestimmen der Häufigkeit des Vorkommens für jeden der m möglichen Schlüsselwerte („Streuen“)
  - Bestimmung der akkumulierten Häufigkeiten in COUNT
  - „Sammeln“ von i = 1 bis m: falls COUNT [i] > 0 wird i-ter Schlüsselwert COUNT [i]-mal ausgegeben
- Beispiel: m=10; Eingabe 4 0 1 1 3 5 6 9 7 3 8 5 (n=12)
- Kosten
  - Keine Duplikate
    a) Streuen $C_1 \cdot n$
    b) Sammeln $C_2 \cdot m$
  - mit Duplikaten: d
    a) Streuen $C_1 \cdot n$
    b) Sammeln C_2 (m+d)



Distribution Sort (2)

- Algorithmus

```
public class SimpleDistributionSort {
  /** Sortiert ein Feld von Werten im Bereich 1 bis m */
  public static void sort (int[] A) {
    int i,j,k;                    // Laufvariablen
    int m = 1000;                 // m mögliche Schlüsselwerte (0 bis 999)
    int n = A.length-1;           // Position 0 des Arr. wird nicht betrachtet
    int[] count = new int[m];     // Zähler für die Anzahl der einzelnen Zeichen
    for (i=0; i<m; i++) count[i] = 0;  // Zähler initialisieren
    for (i=1; i<=n; i++)               // Zählen der Zeichen (Streuphase)
      count[A[i]]++;
    k=0;
    for (i=0; i<m; i++)           // Sammelphase
      for (j=0; j<count[i]; j++)
        A[k++] = i;               // schreibe i count[i] mal in A ein
  }
}
```

Distribution-Sort / Fachverteilen

- Verallgemeinerung: Sortierung von k-Tupeln gemäß lexikographischer Ordnung („Fachverteilen“)
- Lexikographische Ordnung:

  $A = \{a_1, a_2, \ldots, a_n\}$ sei Alphabet mit gegebener Ordnung $a_1 < a_2 < \ldots < a_n$

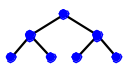
  die sich wie folgt auf A* fortsetzt:

  $v \leq w$ $(v, w \in A^*)$: $\Leftrightarrow$ $(w = v\,u$ mit $u \in A^*)$ oder

  $(v = u\,a_i\,u'$ und $w = u\,a_j\,u''$ mit $u, u', u'' \in A^*$ und $a_i, a_j \in A$ mit $i < j)$

  Die antisymmetrische Relation ”≤” heißt lexikographische Ordnung.
- Sortierung erfolgt in k Schritten:
  - Sortierung nach der letzten Stelle
  - Sortierung nach der vorletzten Stelle

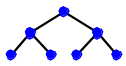
    ...
  - Sortierung nach der ersten Stelle
- Beispiel: Sortierung von Postleitzahlen
  67663, 04425, 80638, 35037, 55128, 04179, 79699, 71672



Distribution-Sort (4)

```
public class  DistributionSort{
  public static void sort (Orderable[] A, int m, int k) {
    for (int i=k; i > 0; i--) {   // läuft über alle Schlüsselpositionen, beginnend mit der letzten
      int[] count = new int[m];
      for (int j=0; j < m; j++) count[j] = 0;            // count initialisieren
      for (int j=1; j < A.length; j++)
        count[key(i, k, A[j])]++; // Bedarf für Fachgröße bestimmen
      for (int j=1; j < m; j++) count[j] += count[j-1];   // Aufsummieren
      for (int j=m-1; j > 0; j--)
        count[j] = count[j-1]+1;  // Beginn der Fächer im Array bestimmen
      count[0] = 1;               // Fach für Objekt j beginnt an Pos. count[j]

     Orderable B = new Orderable[A.length];
      for (int j=1; j < A.length; j++) {  // Einordnen und dabei Fachgrenze
        int z=key(i, k, A[j]);            // anpassen
        B[count[z]++] = A[j];
      }
      System.arraycopy(B, 0, A, 0, A.length); // kopiere B -> A
    }
  }
}
```

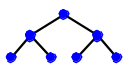
- Aufwand:



Merge-Sort

- anwendbar für internes und externes Sortieren
- Ansatz (Divide-and-Conquer-Strategie)
  - rekursives Zerlegen in jeweils gleich große Teilfolgen
  - Verschmelzen (Mischen) der sortierten Teilfolgen
- Algorithmus (2-Wege-Merge-Sort)

```
public class MergeSort extends SortAlgorithm {
  static void sort (Orderable A[], int l, int r) {
    if (l < r) {
      int m = (l+r+1)/2;    // auf Mitte der Liste zeigen
      sort (A, l, m-1);     // linke Teil-Liste sortieren
      sort (A, m, r);       // rechte Teil-Liste sortieren
      merge(A, l, m, r);    // linken und rechten Teil mischen
    }
  }

  public static void sort (Orderable A[]) {
    sort(A, 1, A.length-1);
  }
}
```

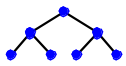



Merge-Sort (2)

- Mischen (Prozedur Merge)
  - pro Teilliste wird Zeiger (Index) auf nächstes Element geführt
  - jeweils kleinstes Element wird in Ergebnisliste übernommen und Indexzeiger fortgeschaltet
  - sobald eine Teilliste „erschöpft“ ist, werden alle verbleibenden Elemente der anderen Teilliste in die Ergebnisliste übernommen
  - lineare Kosten

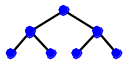
```
static void merge(Orderable A[], int l, int m, int r) {
  Orderable[] B = new Orderable[r-l+1];
  for (int i=0, j=l, k=m; i < B.length; i++)
    if ((k > r) || (j < m) && A[j].less(A[k]))
      B[i] = A[j++];
    else
      B[i] = A[k++];
  for (int i=0; i < B.length; i++) A[l+i] = B[i];
}
```

- Beispiel: Eingabe 7, 4, 15, 3, 11, 17, 12, 8, 18, 14



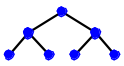
Merge-Sort (3)

- Direktes (reines, nicht-rekursives) 2-Wege-Merge-Sort
  - zunächst werden benachbarte Elemente zu sortierten Teillisten der Länge 2 gemischt
  - fortgesetztes Mischen benachbarter (sortierter) Teillisten
  - Länge sortierter Teillisten verdoppelt sich pro Durchgang
  - Sortierung ist beendet, sobald in einem Durchgang nur noch zwei Teillisten verschmolzen werden
- Beispiel: 7, 4, 15, 3, 11, 17, 12, 8, 18, 14



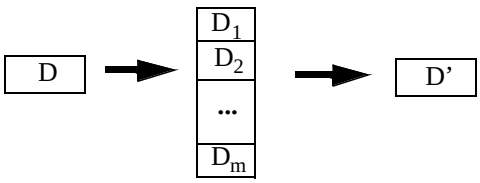
Merge-Sort (4)

- Natürliches 2-Wege-Merge-Sort:
  - statt mit 1-elementigen Teillisten zu beginnen, werden bereits anfangs möglichst lange sortierte Teilfolgen („runs“) verwendet
  - Nutzung einer bereits vorliegenden (natürlichen) Sortierung in der Eingabefolge
- Beispiel: 7, 4, 15, 3, 11, 17, 12, 8, 18, 14

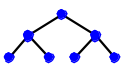


Externes Sortieren

- Anwendung, falls zu sortierende Daten nicht vollständig im Hauptspeicher gehalten werden können
- Hauptziel: Minimierung von Externspeicherzugriffen
- Ansatz
  - Zerlegen der Eingabedatei in m Teildateien, die jeweils im Hauptspeicher sortiert werden können
  - Interne Sortierung und Zwischenspeicherung der Runs
  - Mischen der m Runs (m-Wege-Mischen)

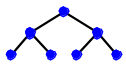


- Bewertung
  - optimale E/A-Kosten (1 Durchgang)
  - setzt Dateispeicherung auf Direktzugriffsmedium (z.B. Magnetplatten) voraus, da ggf. sehr viele temporäre Dateien D_i
  - verfügbarer Hauptspeicher muß wenigstens m+1 Seiten (Blöcke) umfassen
- Restriktiver: Sortieren mit Bändern
  - Ausgeglichenes 2-Wege-Merge-Sort (4 Bänder)
  - Ausgeglichenes k-Wege-Merge-Sort (2k Bänder)



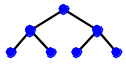
Externes Sortieren (2)

- Ausgeglichenes 2-Wege-Merge-Sort
  - vier Bänder B1, B2, B3, B4; Eingabe sei auf B1; Hauptspeicher fasse r Datensätze
  - Wiederholtes Lesen von r Sätzen von B1, interne Sortierung und abwechselndes Ausschreiben der Runs auf B3 und B4 bis B1 erschöpft ist
  - Mischen der Runs von B3 und B4 (ergibt Runs der Länge 2r) und abwechselndes Schreiben auf B1 und B2
  - Fortgesetztes Mischen und Verteilen bis nur noch 1 Run übrig bleibt
- Beispiel 14, 4, 3, 17, 22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15 (r=4)
  - #Durchgänge:
- Ausgeglichenes k-Wege-Merge-Sort
  - k-Wege-Aufteilen und k-Wege-Mischen mit 2k Bändern
  - #Durchgänge:



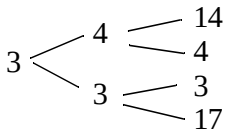
Externes Sortieren (3)

- Mischen bei Mehrwege-Merge-Sort kann durch Auswahlbaum (Turnier-Sortierung oder Heap-Sort) beschleunigt werden:
  Replacement Selection Sort
- Bei k Schlüsseln kann Entfernen des Minimums und Hinzufügen eines neuen Schlüssels in O (log k) Schritten erfolgen
- Auswahlbaum kann auch bei unsortierter Eingabe (initiale Zerlegung) genutzt werden, um Länge der erzeugten Runs zu erhöhen => Sortierung erfordert weniger Durchgänge
  - ausgegebenes (Wurzel-) Element wird im Auswahlbaum durch nächstes Element x aus der Eingabe ersetzt
  - x kann noch im gleichen Run untergebracht werden, sofern x nicht kleiner als das größte schon ausgegebene Element ist
  - Ersetzung erfolgt solange bis alle Schlüssel im Auswahlbaum kleiner sind als der zuletzt ausgegebene (=> neuer Run)
  - im Mittel verdoppelt sich die Run-Länge (auf 2r)

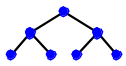


Externes Sortieren (4)

- Beispiel 14, 4, 3, 17, 22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15 (r=4)

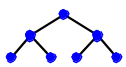


22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15



Zusammenfassung

- Kosten allgemeiner Sortierverfahren wenigstens O (n log n)
- Elementare Sortierverfahren: Kosten O (n^2)
  - einfache Implementierung; ausreichend bei kleinem n
  - gute Lösung: Insertion Sort
- O(n log n)-Sortierverfahren: Heap-Sort, Quick-Sort, Merge-Sort
  - Heap-Sort und Merge-Sort sind Worst-Case-optimal (O (n log n))
  - in Messungen erzielte Quick-Sort in den meisten Fällen die besten Ergebnisse
- Begrenzung der Kosten für Umkopieren durch indirekte Sortierverfahren
- Vorteilhafte Eigenschaften
  - Ausnutzen einer weitgehenden Vorsortierung
  - Stabilität
- lineare Sortierkosten in Spezialfällen erreichbar: Streuen und Sammeln
- Externes Sortieren
  - fortgesetztes Zerlegen und Mischen
  - Mehrwege-Merge-Sort reduziert Externspeicherzugriffe
  - empfehlenswert: Replacement Selection

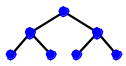


Sortierverfahren: Laufzeitvergleich

- Quelle: Gumm/Sommer: Einführung in die Informatik, 5. Auflage, Oldenbourg-Verlag 2002
- Array mit n ganzen positiven Zahlen; Java-Implementierungen; Messung auf 1,7 GHz Pentium4 PC

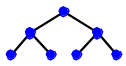
	n=25.000 (vorsortiert)	n=100.000 (vorsortiert)	n=25.000 (unsortiert)	n=100.000 (unsortiert)
BubbleSort	0	0	5,5	89,8
SelectionSort	1,1	19,4	1,1	19,5
InsertionSort	0	0	1,2	19,6

	n=5.000.000 (vorsortiert)	n=20 Mill. (vorsortiert)	n=5.000.000 (unsortiert)	n=20 Mill. (unsortiert)
ShellSort	0,5	2,1	7,4	39,4
QuickSort	0,5	2,2	2,0	8,7
MergeSort	2,5	10,5	3,7	16,1
HeapSort	3,6	15,6	8,3	42,2
DistributionSort	1,06	4,4	1,04	4,3
Bubble / Insertion Sort	0,03	0,13	-	-



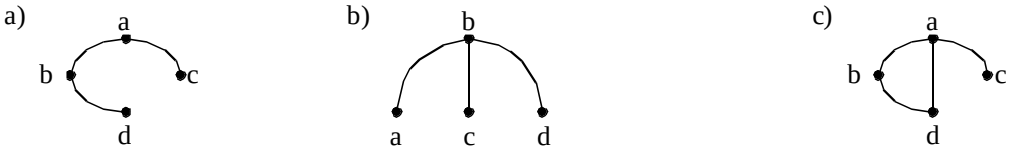
5. Allgemeine Bäume und Binärbäume

- Bäume - Überblick
  - Orientierte Bäume
  - Darstellungsarten
  - Geordnete Bäume
- Binäre Bäume: Begriffe und Definitionen
- Speicherung von binären Bäumen
  - Verkettete Speicherung
  - Feldbaum-Realisierung
  - Sequentielle Speicherung
  - Aufbau von Binärbäumen (Einfügen von Knoten)
- Durchlaufen von binären Bäumen
  - Preorder-, Inorder-, Postorder-Traversierung
  - Rekursive und iterative Version
  - Fädelung



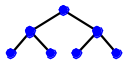
Bäume

- Bäume lassen sich als sehr wichtige Klasse von Graphen auffassen



- Danach ist ein Baum
  - ein azyklischer einfacher, zusammenhängender Graph
  - d.h., er enthält keine Schleifen und Zyklen; zwischen jedem Paar von Knoten besteht höchstens eine Kante
- Def.: Orientierte Bäume: Sei X eine Basis-Datenstruktur. Eine Menge B von Objekten aus X ist ein *orientierter (Wurzel-) Baum*, falls

1. in B ein ausgezeichnetes Element w - Wurzel von B - existiert
2. die Elemente in B - {w} disjunkt zerlegt werden können in B_1, B_2,..., B_m, wobei jedes B_i ebenfalls ein Baum ist.

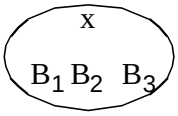


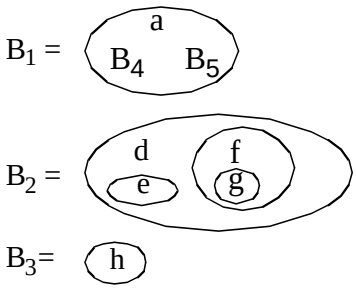
Darstellungsarten für orientierte Bäume

1. Mengendarstellung

Objekte: a, b, c, ...

Bäume: B_1, B_2, B_3, ...





2. Klammerdarstellung

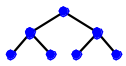
Wurzel = erstes Element innerhalb eines Klammerpaares

$\{x, B_1, B_2, B_3\}$

$\{x, \{a, \{b\}, \{c\}\}, \{d, \{e\}, \{f, \{g\}\}\}, \{h\}\}$

B1 B2 B3

$\{a, \{b\}, \{c\}\} \equiv \{a, \{c\}, \{b\}\}$



Darstellungsarten für orientierte Bäume (2)

3. Rekursives Einrücken

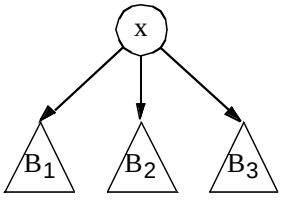
x

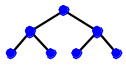
B_1
B_2
B_3

$\Rightarrow$

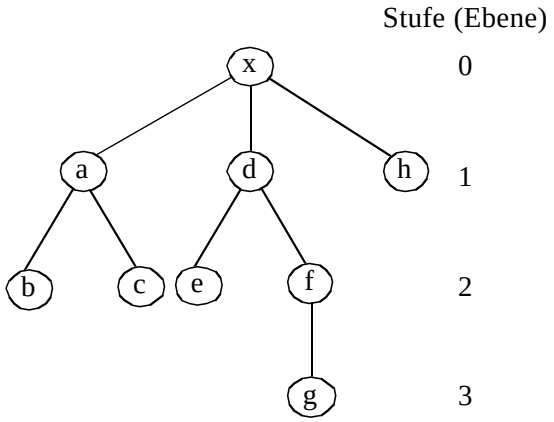
x
a
b
c
d
e
f
g
h

4. Graphendarstellung





Graphendarstellung orientierter Bäume



- Bezeichnungen

Wurzel:

Blätter:

innere Knoten:

Grad K = # Nachfolger von K

Grad (x) =

Grad (g) =

Grad (Baum) = Max (Grad(K_i)) =

Stufe (K_i) = Pfadlänge l von Wurzel nach K_i

Stufe 0:

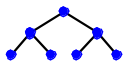
Stufe 1:

Stufe 2:

Stufe 3:

Höhe h =

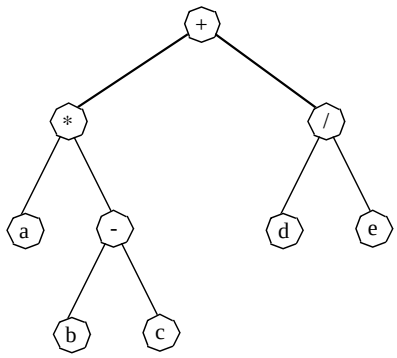
Gewicht w = # der Blätter =



Geordnete Bäume

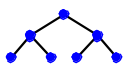
- Def.: Bei einem geordneten Baum bilden die Unterbäume B_i jedes Knotens eine geordnete Menge
- Def.: Eine geordnete Menge von geordneten Bäumen heißt Wald
- Beispiel: Arithmetischer Ausdruck a * (b - c) + d/e

Graphendarstellung



Klammerdarstellung

{+, { *, { a }, {-, { b }, { c } } }, { /, { d }, { e } } }



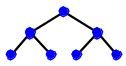
Binärbäume

- Def.: Ein Binärbaum ist eine endliche Menge von Elementen, die entweder leer ist oder ein ausgezeichnetes Element - die Wurzel des Baumes - besitzt und folgende Eigenschaften aufweist:
  - Die verbleibenden Elemente sind in zwei disjunkte Untermengen zerlegt.
  - Jede Untermenge ist selbst wieder ein Binärbaum und heißt linker bzw. rechter Unterbaum des ursprünglichen Baumes
- Formale ADT-Spezifikation

Datentyp BINTREE
Basistyp ELEM

Operationen:

CREATE:		→	BINTREE
EMPTY:	BINTREE	→	{TRUE, FALSE}
BUILD:	BINTREE × ELEM ×BINTREE	→	BINTREE
LEFT:	BINTREE - $\{b_0\}$	→	BINTREE
ROOT:	BINTREE - $\{b_0\}$	→	ELEM
RIGHT:	BINTREE - $\{b_0\}$	→	BINTREE



Axiome:

CREATE	=	b_0;
EMPTY (CREATE)	=	TRUE;

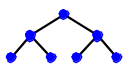
∀ l, r, ∈ BINTREE, ∀ d ∈ ELEM:

EMPTY (BUILD (l, d, r))	=	FALSE;
LEFT (BUILD (l, d, r))	=	l;
ROOT (BUILD (l, d, r))	=	d;
RIGHT (BUILD (l, d, r)	=	r;

- Welche Binärbäume (geordneten Bäume) entstehen durch

BUILD (BUILD (0, b, BUILD (0, d, 0)), a, BUILD (0, c, 0))

BUILD (BUILD (BUILD (0, d, 0), b, 0), a, BUILD (0, c, 0))



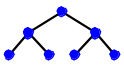
Binärbäume (2)

- **Satz:** Die maximale Anzahl von Knoten eines Binärbaumes

  (1) auf Stufe i ist 2^i, $i \geq 0$

  (2) der Höhe h ist 2^h-1, $h \geq 1$

- **Def.:** Ein vollständiger Binärbaum der Stufe k hat folgende Eigenschaften:
  - Jeder Knoten der Stufe k ist ein Blatt.
  - Jeder Knoten auf einer Stufe < k hat nicht-leere linke und rechte Unterbäume.
- **Def.:** In einem strikten Binärbaum besitzt jeder innere Knoten nicht-leere linke und rechte Unterbäume



Binärbäume (3)

- **Def.:** Ein fast vollständiger Binärbaum ist ein Binärbaum, so daß gilt:

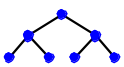
  (1) Jedes Blatt im Baum ist auf Stufe k oder k+1 ($k \geq 0$)

  (2) Jeder Knoten auf Stufe < k hat nicht-leere linke und rechte Teilbäume

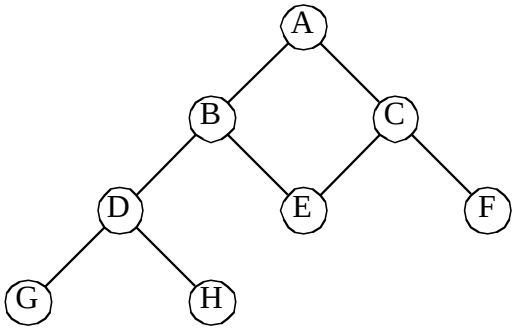
  (3) Falls ein innerer Knoten einen rechten Nachfolger auf Stufe k+1 besitzt, dann ist sein linker Teilbaum vollständig mit Blättern auf Stufe k+1
- **Def.:** Ein ausgeglichener Binärbaum ist ein Binärbaum, so daß gilt:

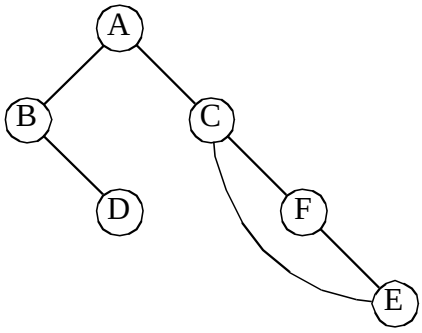
  (1) Jedes Blatt im Baum ist auf Stufe k oder k+1 ($k \geq 0$)

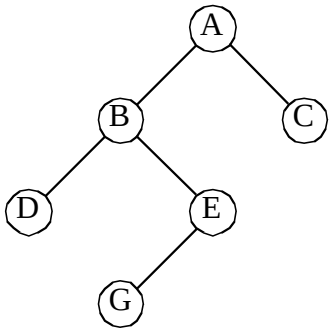
  (2) Jeder Knoten auf Stufe < k hat nicht-leere linke und rechte Teilbäume
- Zwei Binärbäume werden als *ähnlich* bezeichnet, wenn sie dieselbe Struktur besitzen. Sie heißen *äquivalent*, wenn sie ähnlich sind und dieselbe Information enthalten.

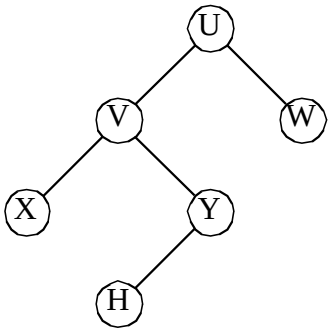


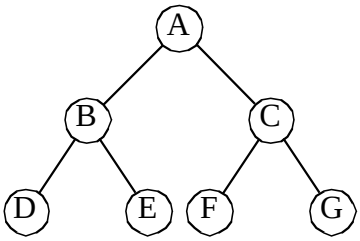
Veranschaulichung der Definitionen

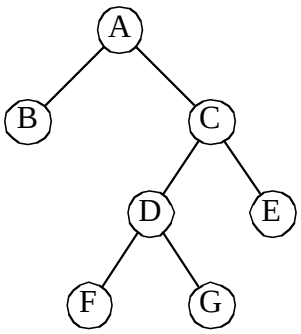


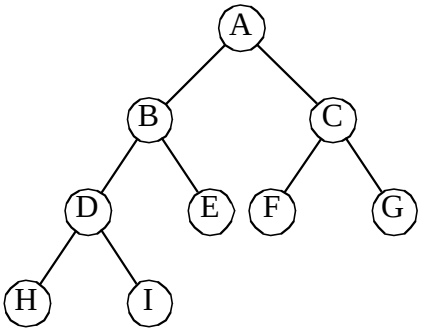


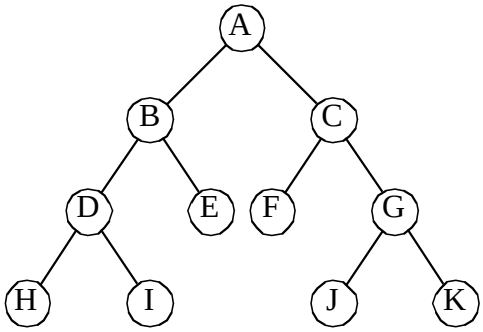






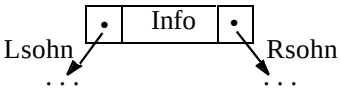






Speicherung von Binärbäumen

1. Verkettete Speicherung



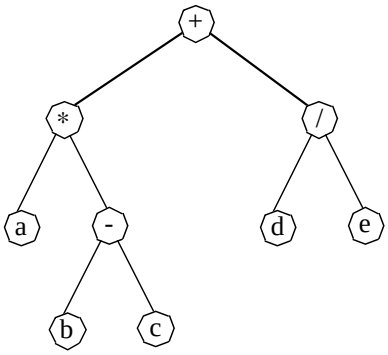
- Freispeicherverwaltung der Struktur wird von der Speicherverwaltung des Programmiersystems übernommen

2. Feldbaum-Realisierung

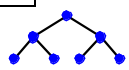
- Simulation einer dynamischen Struktur in einem statischen Feld

Eigenschaften:

- statische Speicherplatzzuordnung
- explizite Freispeicherverwaltung



	Info	Lsohn	Rsohn
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

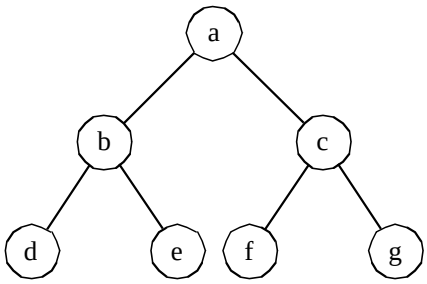


3. Sequentielle Speicherung

Methode kommt **ohne explizite Verweise** aus. Für fast vollständige oder zumindest ausgeglichene Binärbäume bietet sie eine sehr elegante und effiziente Darstellungsform an.

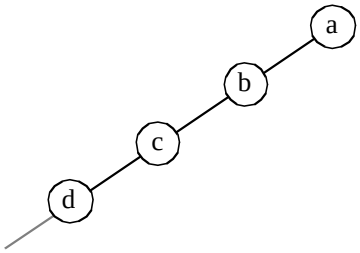
Satz: Ein fast vollständiger Baum mit n Knoten sei sequentiell nach obigem Numerierungsschema gespeichert.
Für jeden Knoten mit Index i, $1 \le i \le n$, gilt:

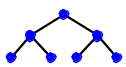
- Vater(i) hat Nummer $\lfloor i/2 \rfloor$ für $i > 1$
- Lsohn(i) hat Nummer 2i für $2i \le n$
- Rsohn(i) hat Nummer 2i+1 für $2i+1 \le n$.



Stufe		Info
0	1	
	2	
	3	
1	4	
	5	
	6	
2	7	

■ Wie sieht der entartete Fall aus?

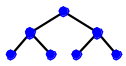




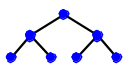
Aufbau von Binärbäumen

- Operationen zum Aufbau eines Binärbaums (Einfügen von Knoten) sowie dem Entfernen von Knoten sind relativ einfach
- Java-Realisierung für gekettete Repräsentation

```
class BinaryNode {
   BinaryNode lChild = null;
   BinaryNode rChild = null;
   Object info = null;

   /** Konstruktor */
   BinaryNode(Object info) { this.info = info; }
 }
```

```
public class BinaryTree {
  private BinaryNode root = null;

  public BinaryTree() { root = null; }

  public BinaryTree(BinaryNode root) { this.root = root; }

  public boolean empty() { return root == null; }

  public void build(BinaryTree lTree, BinaryNode elem, BinaryTree rTree)
  throws TreeException {
    if (elem == null) throw new TreeException("Wurzel ist null!");
    BinaryNode tmpRoot = elem;
    if (lTree == null) tmpRoot.lChild = null;
    else tmpRoot.lChild = lTree.getRoot();
    if (rTree == null) tmpRoot.rChild = null;
    else tmpRoot.rChild = rTree.getRoot();
    root = tmpRoot; }

  public BinaryNode getRoot() { return root; }

  public BinaryTree left() throws TreeException {
    if (empty()) throw new TreeException("Wurzel ist null!");
    return new BinaryTree(root.lChild);
  }

  public BinaryTree right() throws TreeException {
    if (empty()) throw new TreeException("Wurzel ist null!");
    return new BinaryTree(root.rChild);
  }
}
```

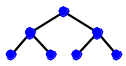



Durchlaufen eines Binärbaums

- **Baumdurchlauf** (Traversierung): Verarbeitung aller Baumknoten gemäß vorgegebener Strukturierung
- Rekursiv anzuwendende Schritte
  1. Verarbeite Wurzel: W
  2. Durchlaufe linken UB: L
  3. Durchlaufe rechten UB: R
- Durchlaufprinzip impliziert sequentielle, lineare Ordnung auf der Menge der Knoten
- 6 Möglichkeiten:
- 3 Strategien verbleiben aufgrund Konvention: linker UB vor rechtem UB

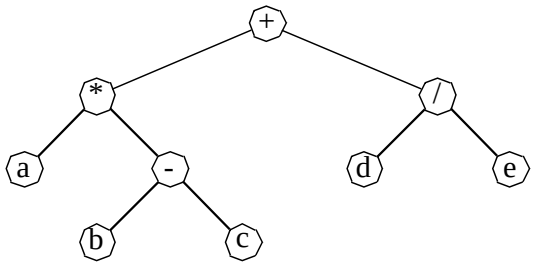
1	2	3	4	5	6
W	L	L	W	R	R
L	W	R	R	W	L
R	R	W	L	L	W

1. Vorordnung (preorder): WLR
2. Zwischenordnung (inorder): LWR
3. Nachordnung (postorder): LRW



Durchlaufmöglichkeiten

Referenzbeispiel

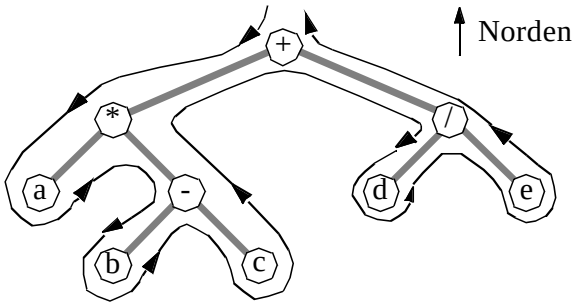


WLR:

LWR:

LRW:

Anschauliche Darstellung

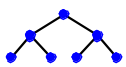


Ausgabe bei Passage des Turms (Knoten)

WLR: in Richtung Süden

LWR: an seiner Südseite

LRW: in Richtung Norden



Rekursive und iterative Realisierung

- Rekursive Version für Inorder-Traversierung (LWR)

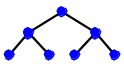
```
public void printInOrder (BinaryNode current) {
  if (current != null) {
    printInOrder(current.lChild);
    System.out.println(current.info);
    printInOrder(current.rChild);
  }
}
```

- **Iterative Version**: LWR
  - Ziel: effizientere Ausführung durch eigene Stapelverwaltung
  - Vorgehensweise: Nimm, solange wie möglich, linke Abzweigung und speichere den zurückgelegten Weg auf einem Stapel.

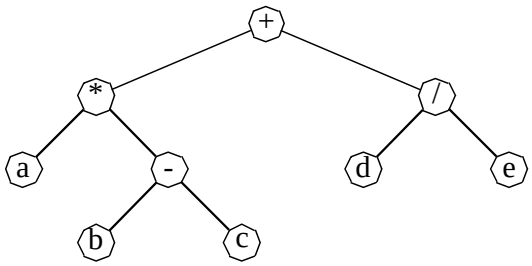
```
Aktion 1:   stack.push(current);
            current = current.lChild;
```

  - Wenn es links nicht mehr weitergeht, wird der oberste Knoten des Stapels ausgegeben und vom Stapel entfernt. Der Durchlauf wird mit dem rechten Unterbaum des entfernten Knotens fortgesetzt.

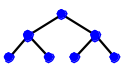
```
Aktion 2:   System.out.println(((BinaryNode) stack.top()).info);
            current = ((BinaryNode) stack.top()).rChild;
            stack.pop();
```

Iterative Version: Durchlaufbeispiel



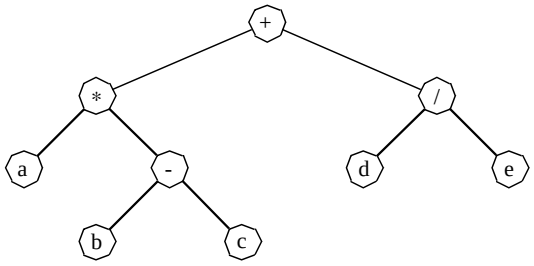
Stapel S	current	Aktion	Ausgabe



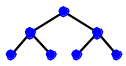
Gefädelte Binärbäume

- Weitere Verbesserung von iterativen Durchlaufalgorithmen
- Methode benutzt einen “Faden”, der die Baumknoten in der Folge der Durchlaufordnung verknüpft.
  - Zwei Typen von Fäden
  - *Rechtsfaden* verbindet jeden Knoten mit seinem Nachfolgerknoten in Durchlaufordnung
  - *Linksfaden* stellt Verbindung zum Vorgängerknoten in Durchlaufordnung her
- Lösung 1. Explizite Speicherung von 2 Fäden

```
class ThreadedBinNode {
  ThreadedBinNode lChild = null;
  ThreadedBinNode rChild = null;
  Object info = null;
  ThreadedBinNode lThread = null;
  ThreadedBinNode rThread = null;
  /** Konstruktor */
  ThreadedBinNode(Object info) {
    this.info = info; } }
```



- Beispiel: Zwischenordnung

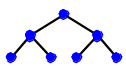


Gefädelte Binärbäume (2)

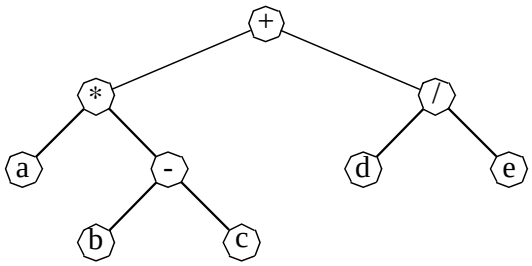
- Lösung 2. Vermeidung von Redundanz

  Eine zweite Art der Fädelung kommt ohne zusätzliche Zeiger aus und erfordert daher geringeren Speicherplatzaufwand. Die Wartungs- und Durchlauf-Algorithmen werden lediglich geringfügig komplexer.
- Beobachtung 1: Binärbaum mit n Knoten hat n+1 freie Zeiger (null)
- Beobachtung 2: für die Zwischenordnung können Fadenzeiger in inneren Knoten durch Folgen von Baumzeigern ersetzt werden
- Idee: Benutze freie Zeiger und Baumzeiger für Fädelung
  - pro Knoten zusätzliche Boolesche Variablen Lfaden, Rfaden:

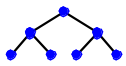
```
class OptThreadedBinNode {
    OptThreadedBinNode lChild = null;
    OptThreadedBinNode rChild = null;
    Object info = null;
    boolean lThread = null;  // true, wenn lChild Fadenzeiger
    boolean rThread = null;  // true, wenn rChild Fadenzeiger
    /** Konstruktor */
    OptThreadedBinNode(Object info) { this.info = info; } }
```

- Prozedur für Traversierung in Zwischenordnung mit optimierter Fädelung

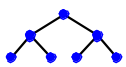



```
public class ThreadedBinTree {
   private OptThreadedBinNode root = null;
   ...
// Baumdurchlauf iterativ über Fädelung
   public void printInOrder () {
     OptThreadedBinNode current = null;
     OptThreadedBinNode previous = null;
    if (root == null) return;  // Baum leer
     current = root;
   do {
       while ((! current.lThread) && (current.lChild != null))
        current = current.lChild;  // verzweige nach links so lange wie mgl.
       System.out.println(current.info);
       previous = current;
       current = current.rChild;
       while ((previous.rThread) && (current != null)) {
         System.out.println(current.info);
         previous = current;
         current = current.rChild;
       }
     } while (current != null);
   }
 }
```

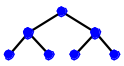
Zusammenfassung

- Definitionen
  - Baum, orientierter Baum (Wurzel-Baum), geordneter Baum, Binärbaum
  - vollständiger, fast vollständiger, strikter, ausgeglichener, ähnlicher, äquivalenter Binärbaum
  - Höhe, Grad, Stufe / Pfadlänge, Gewicht
- Speicherung von Binärbäumen
  - verkettete Speicherung
  - Feldbaum-Realisierung
  - sequentielle Speicherung
- Baum-Traversierung
  - Preorder (WLR): Vorordnung
  - Inorder (LWR): Zwischenordnung
  - Postorder (LRW): Nachordnung
- Gefädelte Binärbäume: Unterstützung der (iterativen) Baum-Traversierung durch Links/Rechts-Zeiger auf Vorgänger/Nachfolger in Traversierungsreihenfolge



6. Binäre Suchbäume

- Natürliche binäre Suchbäume
  - Begriffe und Definitionen
  - Grundoperationen: Einfügen, sequentielle Suche, direkte Suche, Löschen
  - Bestimmung der mittleren Zugriffskosten
- Balancierte Binärbäume
- AVL-Baum
  - Einfügen mit Rotationstypen
  - Löschen mit Rotationstypen
  - Höhe von AVL-Bäumen
- Gewichtsbalancierte Binärbäume
- Positionssuche mit balancierten Bäumen (Lösung des Auswahlproblems)



Binäre Suchbäume

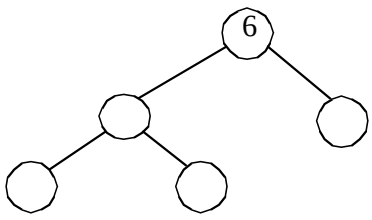
- Def.: Ein natürlicher binärer Suchbaum B ist ein Binärbaum;
  er ist entweder leer oder jeder Knoten in B enthält einen Schlüssel und:

  (1) alle Schlüssel im linken Unterbaum von B sind kleiner als der Schlüssel in der Wurzel von B

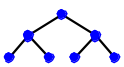
  (2) alle Schlüssel im rechten Unterbaum von B sind größer als der Schlüssel in der Wurzel von B

  (3) die linken und rechten Unterbäume von B sind auch binäre Suchbäume.

Beispiel:

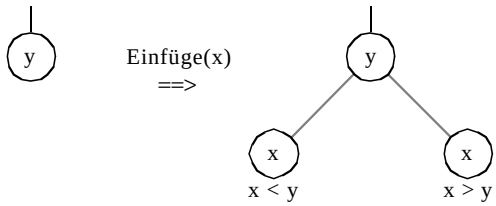


- 4 Grundoperationen:
  - Einfügen
  - direkte Suche
  - sequentielle Suche
  - Löschen



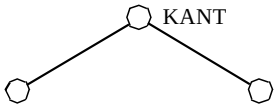
Einfügen in binären Suchbäumen

- Neue Knoten werden immer als Blätter eingefügt
- Suche der Einfügeposition:
- Aussehen des Baumes wird durch die Folge der Einfügungen bestimmt: *reihenfolgeabhängige* Struktur
- n Schlüssel erlauben n! verschiedene Schlüsselfolgen



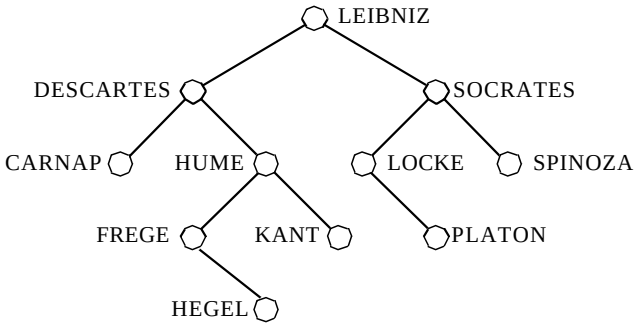
Einfügereihenfolge 1:

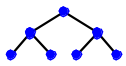
KANT, LEIBNIZ, HEGEL, HUME, LOCKE, SOCRATES, SPINOZA, DESCARTES, CARNAP, FREGE, PLATON



Einfügereihenfolge 2:

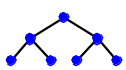
LEIBNIZ, DESCARTES, CARNAP, HUME, SOCRATES, FREGE, LOCKE, KANT, HEGEL, PLATON, SPINOZA





Einfügen in binären Suchbäumen (2)

```
class BinaryNode {
  BinaryNode lChild = null;
  BinaryNode rChild = null;
  Orderable key = null;
   /** Konstruktor */
  BinaryNode(Orderable key) { this.key = key; } }

public class BinarySearchTree {
  private BinaryNode root = null;
  ...
  public void insert(Orderable key) throws TreeException {
    root = insert(root, key); }

  protected BinaryNode insert(BinaryNode node, Orderable key)
  throws TreeException {
    if (node == null) return new BinaryNode(key);
      else
        if (key.less(node.key))
          node.lChild = insert(node.lChild, key);
        else
          if (key.greater(node.key))
            node.rChild = insert(node.rChild, key);
          else
            throw new TreeException("Schluessel schon vorhanden!");
    return node; }
  ...
}
```

Suche in binären Suchbäumen

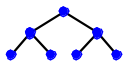
1. Sequentielle Suche

Einsatz eines Durchlauf-Algorithmus (Zwischenordnung)

2. Direkte Suche: Vorgehensweise wie bei Suche nach Einfügeposition

- Suchen eines Knotens (rekursive Version):

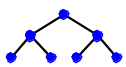
```
/** Rekursive Suche eines Schluessels im Baum */
public boolean searchRec (Orderable key) {
  return searchRec(root, key);
}

/** Rekursive Suche eines Schluessels im Teilbaum */
protected boolean searchRec (BinaryNode node, Orderable key) {
  if (node == null) return false;        // nicht gefunden
  if (key.less(node.key))
    return searchRec(node.lChild, key);  // suche im linken Teilbaum
  if (key.greater(node.key))
    return searchRec(node.rChild, key);  // suche im rechten Teilbaum
  return true;                           // gefunden
}
```

Suchen (2)

- Suchen (iterative Version):

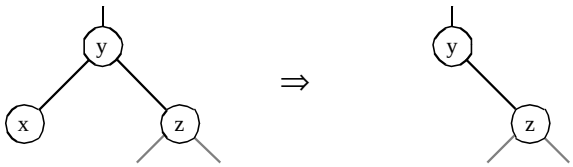
```
public boolean searchIter (Orderable key) {
  BinaryNode node = root;
  do {
    if (node == null) return false;  // nicht gefunden
    if (key.less(node.key))
      node = node.lChild;            // suche im linken Teilbaum
    else if (key.greater(node.key))
      node = node.rChild;            // suche im rechten Teilbaum
    else return true;                // gefunden
  } while (true);
}
```

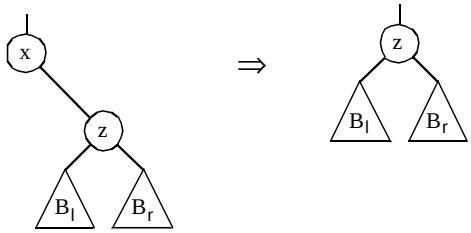
Löschen in binären Suchbäumen

- Löschen ist am kompliziertesten

Fall 1: x ist Blatt

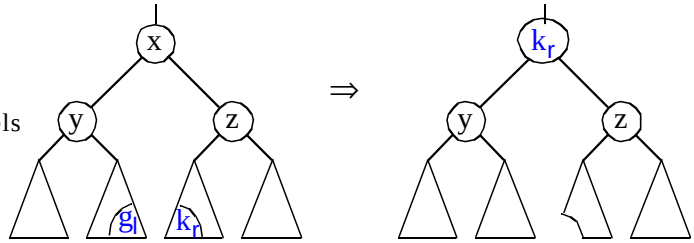


Fall 2/3: x hat leeren linken/rechten Unterbaum

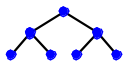


Fall 4: x hat zwei nicht-leere Unterbäume

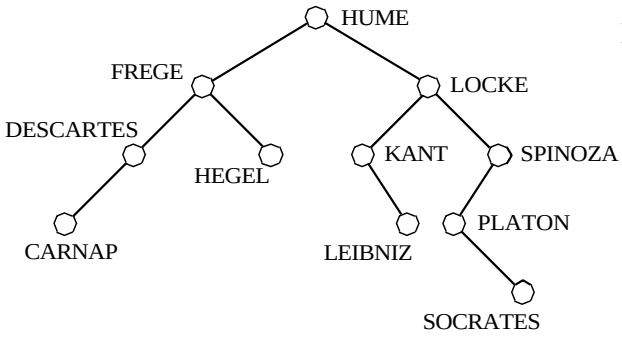
Heranziehen des größten Schlüssels im linken Unterbaum (g_l) oder des kleinsten Schlüssels im rechten Unterbaum (k_r)



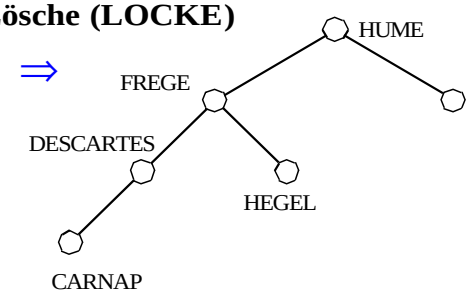
- Alternative: Jeder zu löschende Knoten wird speziell markiert; bei Such- und Einfügevorgängen wird er gesondert behandelt



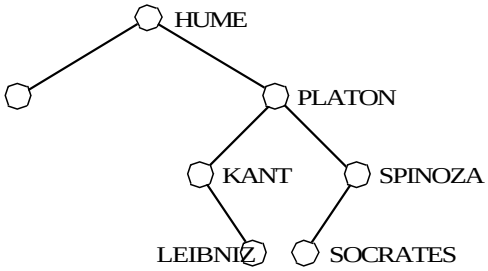
Löschen in binären Suchbäumen (2)



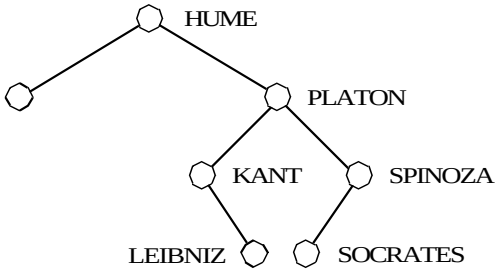
Lösche (LOCKE)

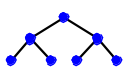


Lösche (DESCARTES)



Lösche (FREGE)





Binäre Suchbäume: Zugriffskosten

- Kostenmaß: Anzahl der aufgesuchten Knoten bzw. Anzahl der benötigten Suchschritte oder Schlüsselvergleiche.
- Kosten der Grundoperationen
  - sequentielle Suche:
  - Einfügen, Löschen, direkte Suche
- Bestimmung der mittleren Zugriffskosten (direkte Suchkosten)
  - Mittlere Zugriffskosten $\bar{z}$ eines Baumes B erhält man durch Berechnung seiner gesamten Pfadlänge PL als Summe der Längen der Pfade von der Wurzel bis zu jedem Knoten K_i.

$$PL(B) = \sum_{i=1}^{n} Stufe(K_i)$$

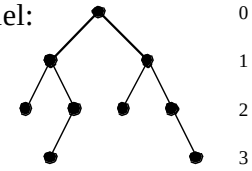
  - mit n_i = Zahl der Knoten auf Stufe i gilt

$$PL(B) = \sum_{i=0}^{h-1} i \cdot n_i \quad \text{und} \quad \sum_{i=0}^{h-1} n_i = n \quad = \text{gesamte Knotenzahl}$$

  - Die mittlere Pfadlänge ergibt sich zu $p = PL / n$
  - Da bei jedem Zugriff noch auf die Wurzel zugegriffen werden muß, erhält man

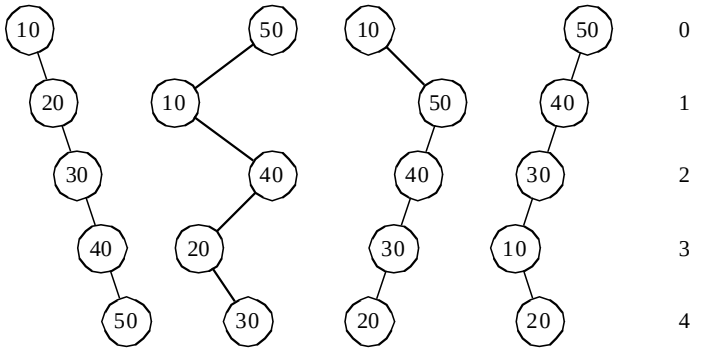
$$\bar{z} = p + 1 = \frac{1}{n} \cdot \sum_{i=0}^{h-1} (i+1) \cdot n_i$$

Beispiel:



Binäre Suchbäume: Zugriffskosten (2)

- Maximale Zugriffskosten
  - Die längsten Suchpfade und damit die maximalen Zugriffskosten ergeben sich, wenn der binäre Suchbaum zu einer linearen Liste entartet



  - Höhe: $h = l_{max} + 1 = n$
  - Maximale mittlere Zugriffskosten:

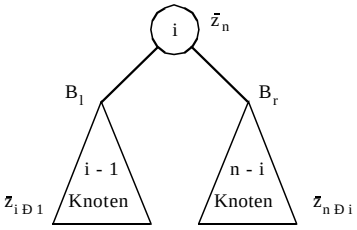
$$\bar{z}_{max} = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (i+1) \cdot 1 = n \cdot \frac{(n+1)}{2n} = \frac{(n+1)}{2} = O(n)$$

- Minimale (mittlere) Zugriffskosten: können in einer fast vollständigen oder ausgeglichenen Baumstruktur erwartet werden
  - Gesamtzahl der Knoten: $2^{h-1} - 1 < n \leq 2^h - 1$
  - Höhe $h = \lfloor \log_2 n \rfloor + 1$
  - Minimale mittlere Zugriffskosten: $\bar{z}_{min} \approx \log_2 n - 1$

Binäre Suchbäume: Zugriffskosten (3)

- Durchschnittliche Zugriffskosten
  - Extremfälle der mittleren Zugriffskosten sind wenig aussagekräftig:
  - Wie groß sind $\bar{z}_{min}$ und $\bar{z}_{max}$ bei n= 10, 10^3, 10^6, . . . ?
  - Differenz der mittleren zu den minimalen Zugriffskosten ist ein Maß für Dringlichkeit von Balancierungstechniken

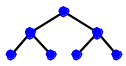
- Bestimmung der mittleren Zugriffskosten
  - n verschiedene Schlüssel mit den Werten 1, 2, ..., n seien in zufälliger Reihenfolge gegeben. Die Wahrscheinlichkeit, daß der erste Schlüssel den Wert i besitzt, ist 1/n (Annahme: gleiche Zugriffswahrscheinlichkeit auf alle Knoten)



  - Für den Baum mit i als Wurzel erhalten wir

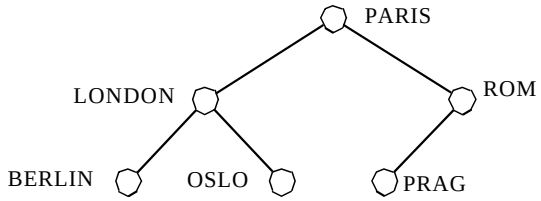
$$\bar{z}_n(i) = \frac{1}{n} \cdot ((\bar{z}_{i-1} + 1) \cdot (i-1) + 1 + (\bar{z}_{n-i} + 1) \cdot (n-i))$$

  - Die Rekursionsgleichung läßt sich in nicht-rekursiver, geschlossener Form mit Hilfe der harmonischen Funktion $H_n = \sum_{i=1}^{n} \frac{1}{i}$ darstellen.
  - Es ergibt sich $\bar{z}_n = 2 \cdot \frac{(n+1)}{n} \cdot H_n - 3 = 2\ln(n) - c.$
  - Relative Mehrkosten: $\frac{\bar{z}_n}{\bar{z}_{min}} = \frac{2\ln(n) - c}{\log_2(n) - 1} \sim \frac{2\ln(n) - c}{\log_2(n)} = 2\ln(2) = 1{,}386\ldots$

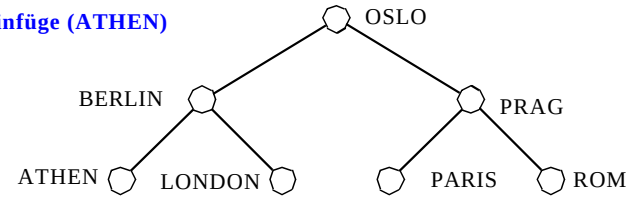


Balancierte Binärbäume

- Der ausgeglichene binäre Suchbaum verursacht für alle Grundoperationen die geringsten Kosten
- **Perfekte Balancierung zu jeder Zeit** kommt jedoch sehr teuer.



Einfüge (ATHEN)

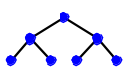


  - In welchem Maße sollen Strukturabweichungen bei Einfügungen und Löschungen toleriert werden ?

- Balancierte Bäume
  - Ziel: schneller direkten Zugriff mit $\bar{z}_{max} \sim O(\log_2 n)$ sowie Einfüge- und Löschoperationen mit logarithmischen Aufwand
  - Heuristik: für jeden Knoten im Baum soll die Anzahl der Knoten in jedem seiner beiden Unterbäume möglichst gleich gehalten werden
  - Zwei unterschiedliche Vorgehensweisen:

    (1) die zulässige Höhendifferenz der beiden Unterbäume ist beschränkt (=> höhenbalancierte Bäume)

    (2) das Verhältnis der Knotengewichte der beiden Unterbäume erfüllt gewisse Bedingungen (=> gewichtsbalancierte Bäume)

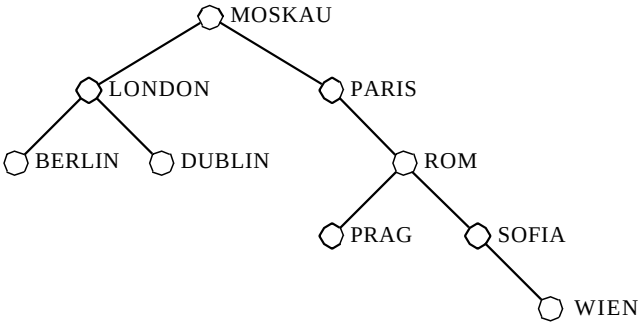


k-balancierter Binärbaum

- Def.: Seien $B_l(x)$ und $B_r(x)$ die linken und rechten Unterbäume eines Knotens x. Weiterhin sei h (B) die Höhe eines Baumes B. Ein k-balancierter Binärbaum ist entweder leer oder es ist ein Baum, bei dem für jeden Knoten x gilt: $|h(B_l(x)) - h(B_r(x))| \leq \mathbf{k}$

  k läßt sich als Maß für die zulässige Entartung im Vergleich zur ausgeglichenen Baumstruktur auffassen

- Prinzip

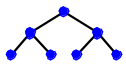


$|h(B_l(SOFIA)) - h(B_r(SOFIA))| =$

$|h(B_l(ROM)) - h(B_r(ROM))| =$

$|h(B_l(PARIS)) - h(B_r(PARIS))| =$

$|h(B_l(MOSKAU)) - h(B_r(MOSKAU))| =$

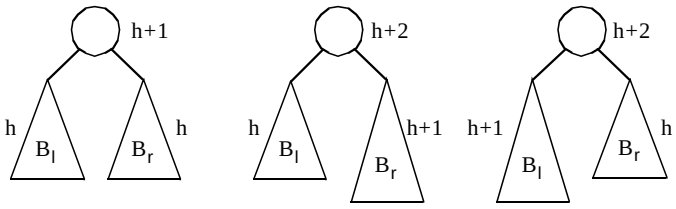


AVL-Baum

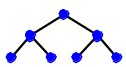
- benannt nach russischen Mathematikern: Adelson-Velski und Landis
- **Def.:** Ein 1-balancierter Binärbaum heißt AVL-Baum

  -> Balancierungskriterium: $|h(B_l(x)) - h(B_r(x))| \leq 1$

- Konstruktionsprinzip:
  - B_l und B_r seien AVL-Bäume der Höhe h und h+1. Dann sind die nachfolgend dargestellten Bäume auch AVL-Bäume:



- Suchoperationen wie für allgemeine binäre Suchbäume



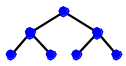
AVL-Baum: Wartungsalgorithmen

- Wann und wo ist das AVL-Kriterium beim Einfügen verletzt ?
  - Es kann sich nur die Höhe von solchen Unterbäumen verändert haben, deren Wurzeln auf dem Suchpfad von der Wurzel des Baumes zum neu eingefügten Blatt liegen
  - Reorganisationsoperationen lassen sich lokal begrenzen; es sind höchstens h Knoten betroffen
- Def.: Der Balancierungsfaktor BF(x) eines Knotens x ergibt sich zu

$$BF(x) = h(B_l(x)) - h(B_r(x)).$$

- Knotendefinition

```
class AVLNode {
  int BF = 0;
  AVLNode lChild = null;
  AVLNode rChild = null;
  Orderable key = null;
  /** Konstruktor */
  AVLNode(Orderable key) { this.key = key; }
}
```

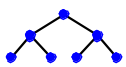
Einfügen in AVL-Bäumen

- Sobald ein BF(x) durch eine Einfügung verletzt wird, muß eine Rebalancierung des Baumes durch sog. Rotationen durchgeführt werden.
  - Ausgangspunkt der Rotation ist der näheste Vater des neu eingefügten Knotens mit `BF = ∓2`.
  - Dieser Knoten dient zur Bestimmung des Rotationstyps. Er wird durch die von diesem Knoten ausgehende Kantenfolge auf dem Pfad zum neu eingefügten Knoten festgelegt.
- Rotationstypen

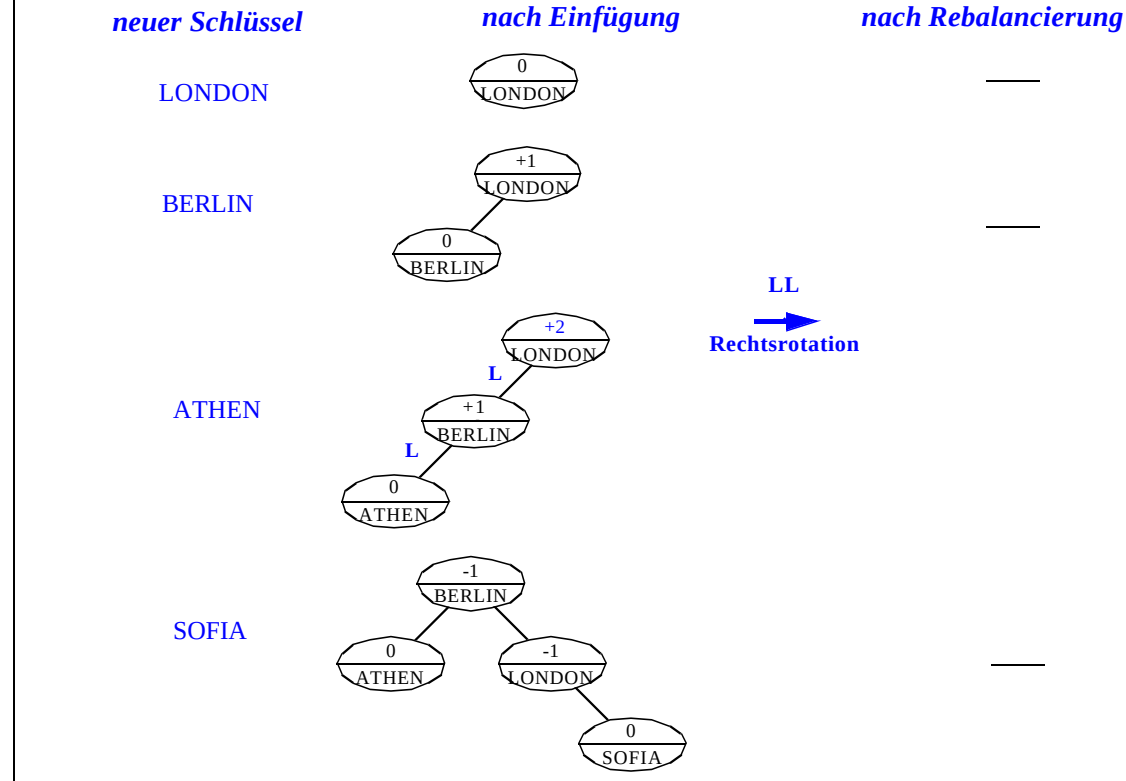
  Es treten vier verschiedene Rotationstypen auf. Der neu einzufügende Knoten sei X. Y sei der bezüglich der Rotation kritische Knoten - der näheste Vater von X mit `BF = ∓2` . Dann bedeutet:

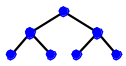
  - RR: X wird im rechten Unterbaum des rechten Unterbaums von Y eingefügt (Linksrotation)
  - LL: X wird im linken Unterbaum des linken Unterbaums von Y eingefügt (Rechtsrotation)
  - RL: X wird im linken Unterbaum des rechten Unterbaums von Y eingefügt (Doppelrotation)
  - LR: X wird im rechten Unterbaum des linken Unterbaums von Y eingefügt (Doppelrotation)

  Die Typen LL und RR sowie LR und RL sind symmetrisch zueinander.

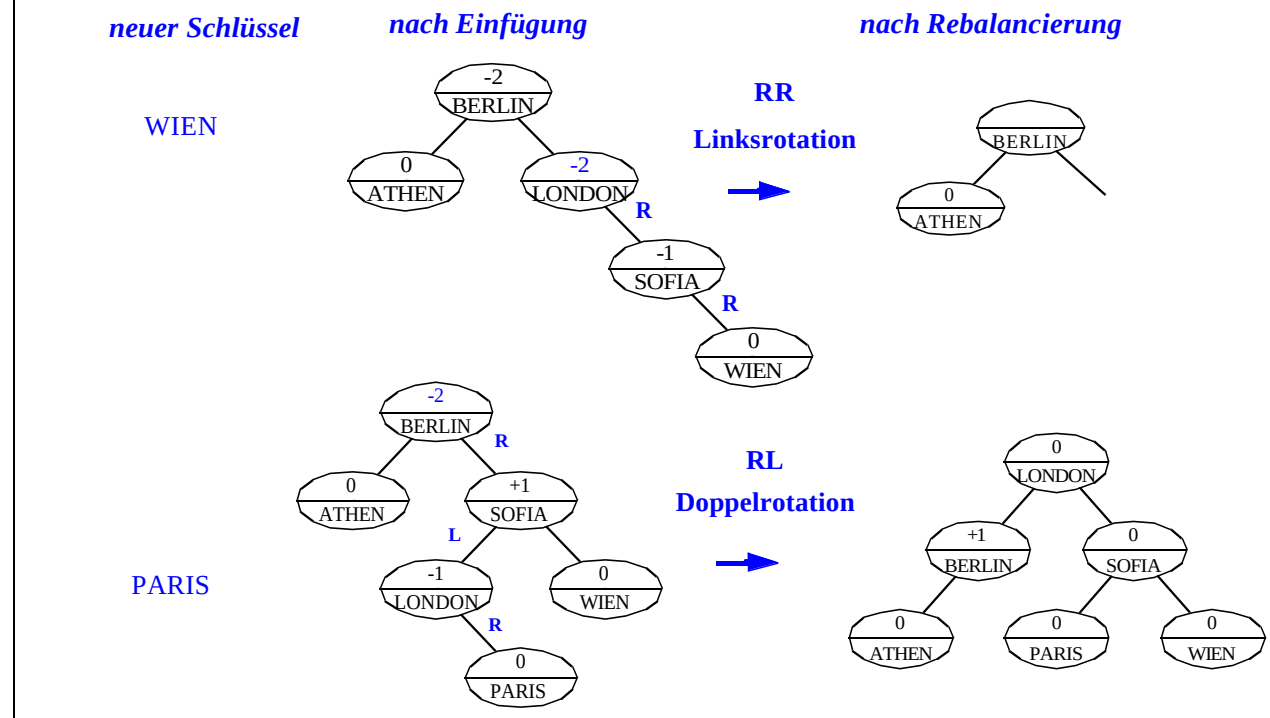


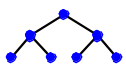
Einfügen in AVL-Bäumen (2)



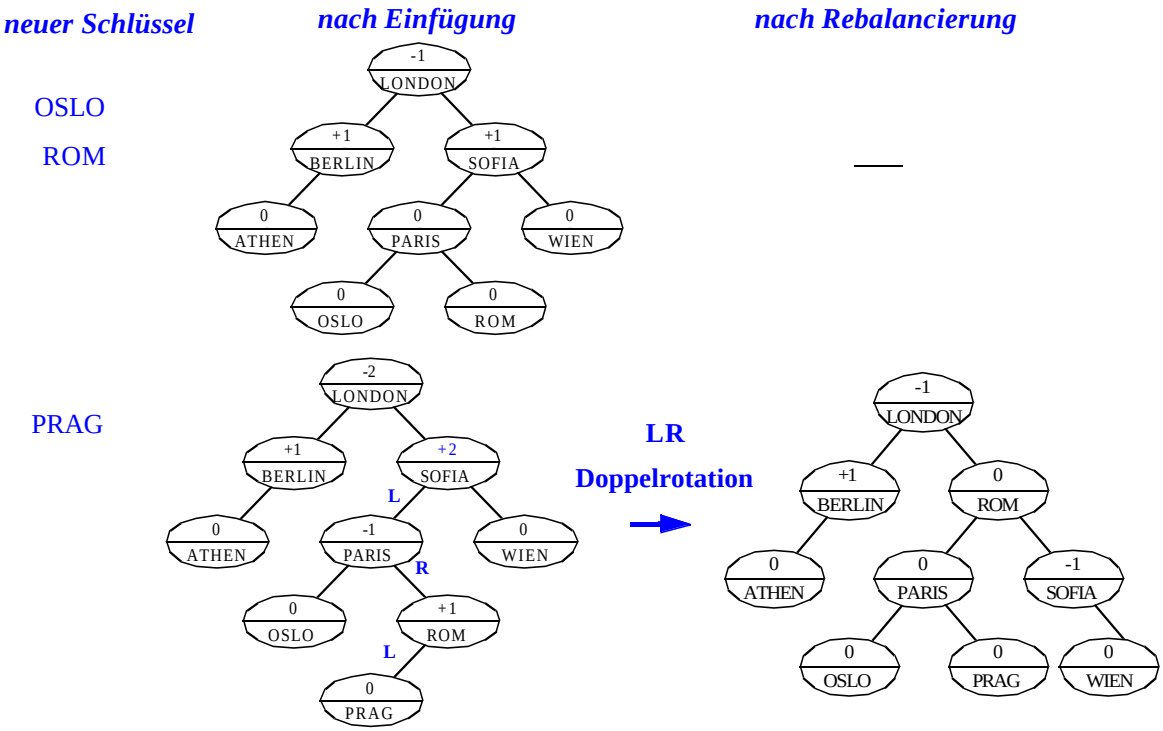


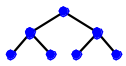
Einfügen in AVL-Bäumen (3)



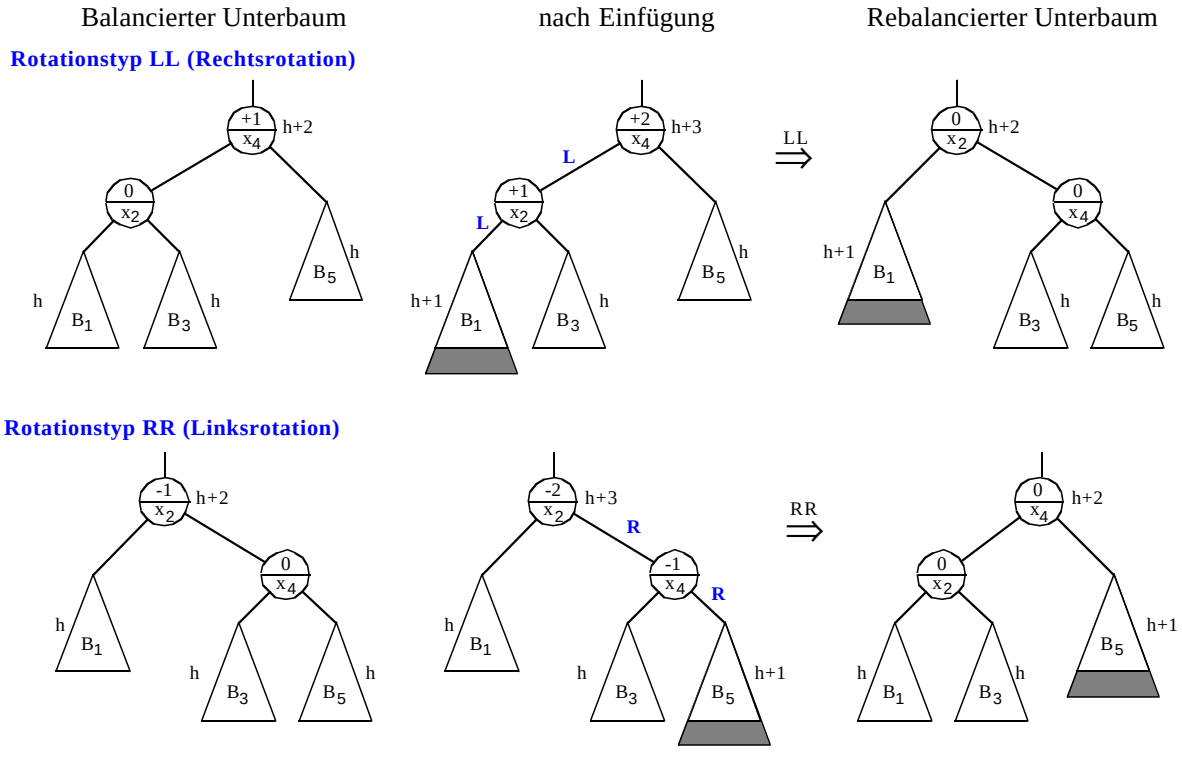


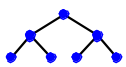
Einfügen in AVL-Bäumen (3)





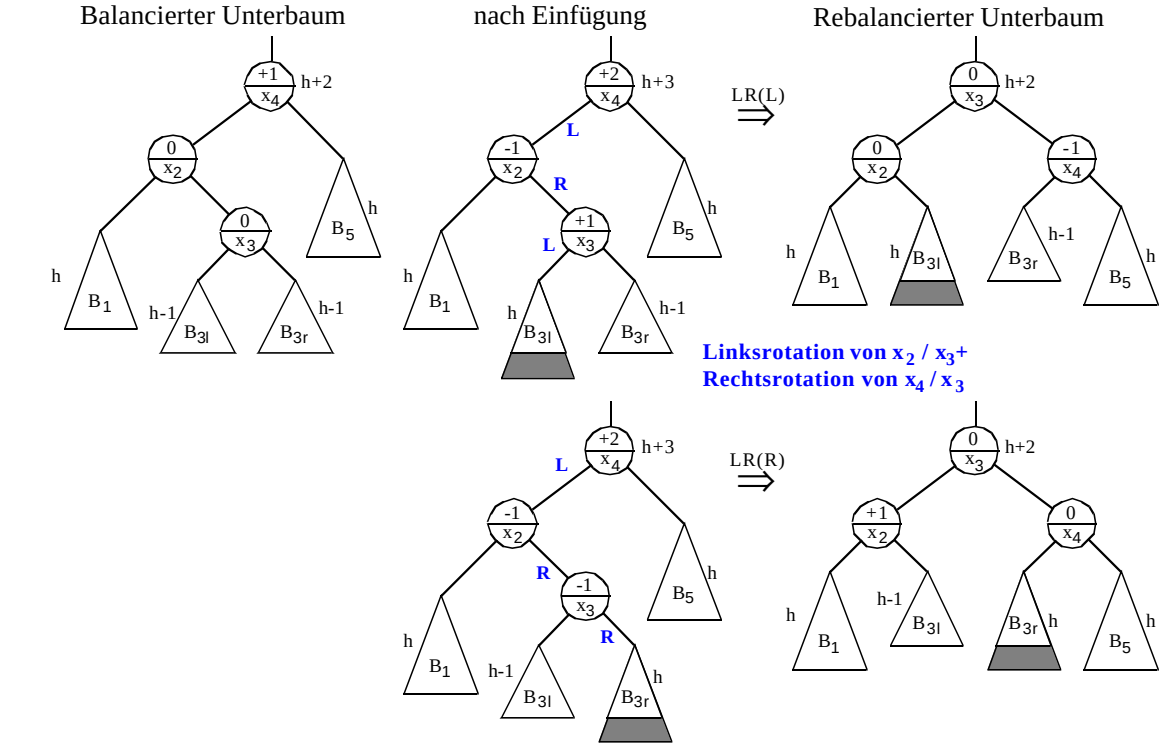
Einfügen in AVL-Bäumen (4)





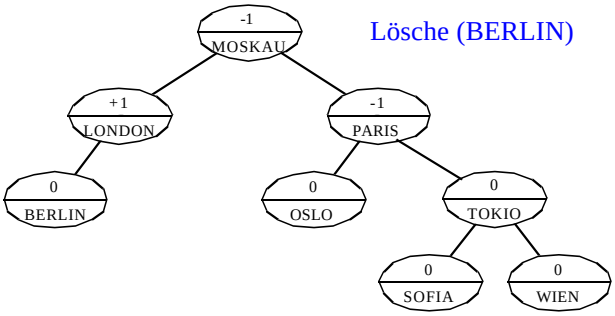
Einfügen in AVL-Bäumen (5)

Rotationstyp LR (RL ist symmetrisch)

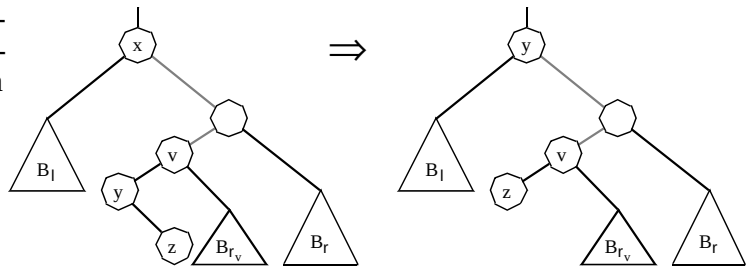


Löschen in AVL-Bäumen

- Löschen eines Blattes bzw. eines Randknotens (max. 1 Sohn)
  - Höhenreduzierung ändert Balancierungsfaktoren der Vaterknoten
  - Rebalancierung für UB der Vorgängerknoten mit BF = +/- 2
  - ggf. fortgesetzte Rebalancierung (nur möglich für Knoten mit BF = +/- 2 auf dem Weg vom zu löschenden Element zur Wurzel)



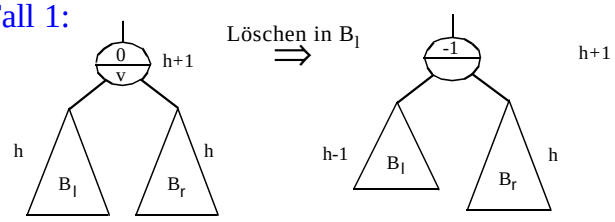
- Löschen eines Knotens (Schlüssel x) mit 2 Söhnen kann auf Löschen für Blatt/Randknoten zurückgeführt werden
  - x wird ersetzt durch kleinsten Schlüssel y im rechten Unterbaumbaum von x (oder größten Schlüssel im linken Unterbaum)
  - führt zur Änderung des Balancierungsfaktors für v (Höhe des linken Unterbaums von v hat sich um 1 reduziert)



Löschen in AVL-Bäumen (2)

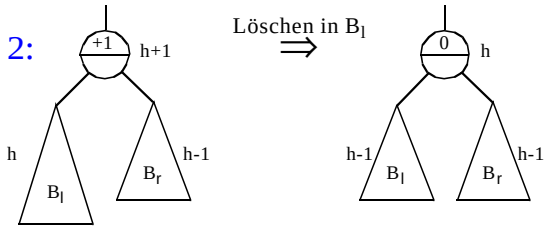
Bis auf Symmetrie treten nur 3 Fälle auf:

Fall 1:



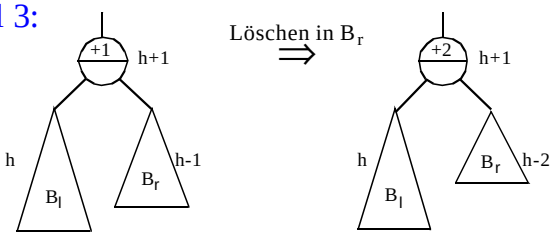
In diesem Fall pflanzt sich Höhenerniedrigung nicht fort, da in der Wurzel das AVL-Kriterium erfüllt bleibt.
-> kein Rebalancieren erforderlich.

Fall 2:

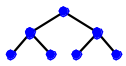


Die Höhenerniedrigung von B_l pflanzt sich hier zur Wurzel hin fort. Sie kann auf diesem Pfad eine Rebalancierung auslösen.

Fall 3:

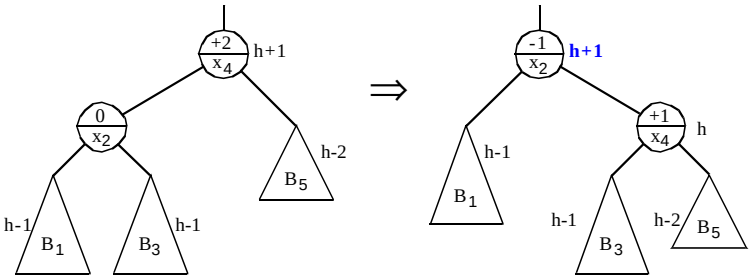


Für die Behandlung dieser Situation ist der linke Unterbaum B_l in größerem Detail zu betrachten. Dabei ergeben sich die 3 Unterfälle:



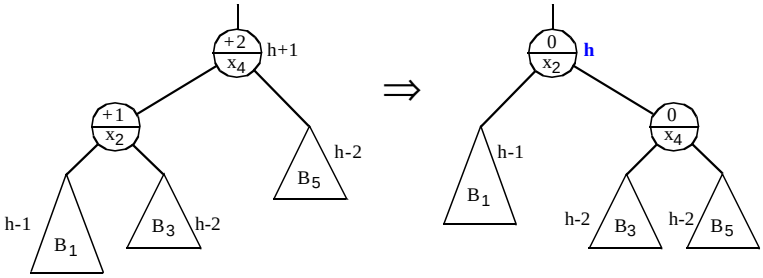
Löschen in AVL-Bäumen (3)

Fall 3a:

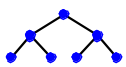


- Rechtsrotation führt zur Erfüllung des AVL-Kriteriums
- Unterbaum behält ursprüngliche Höhe
- keine weiteren Rebalancierungen erforderlich

Fall 3b:

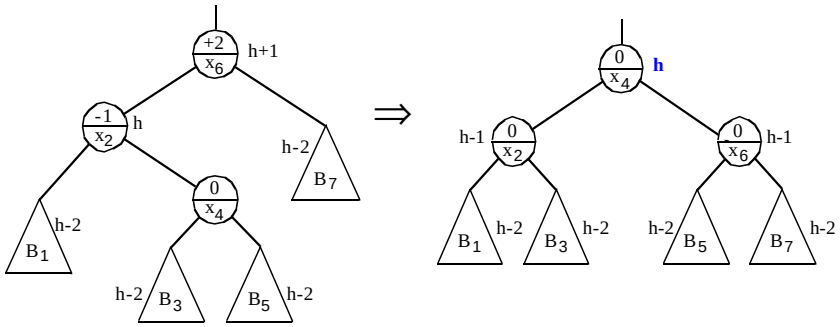


- Rechtsrotation reduziert Höhe des gesamten UB von h+1 nach h
- Höhenreduzierung pflanzt sich auf dem Pfad zur Wurzel hin fort und kann zu weiteren Rebalancierungen führen.

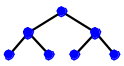


Löschen in AVL-Bäumen (5)

Fall 3c:



- Doppelrotation (-> Höhenerniedrigung)
- ggf. fortgesetzte Rebalancierungen



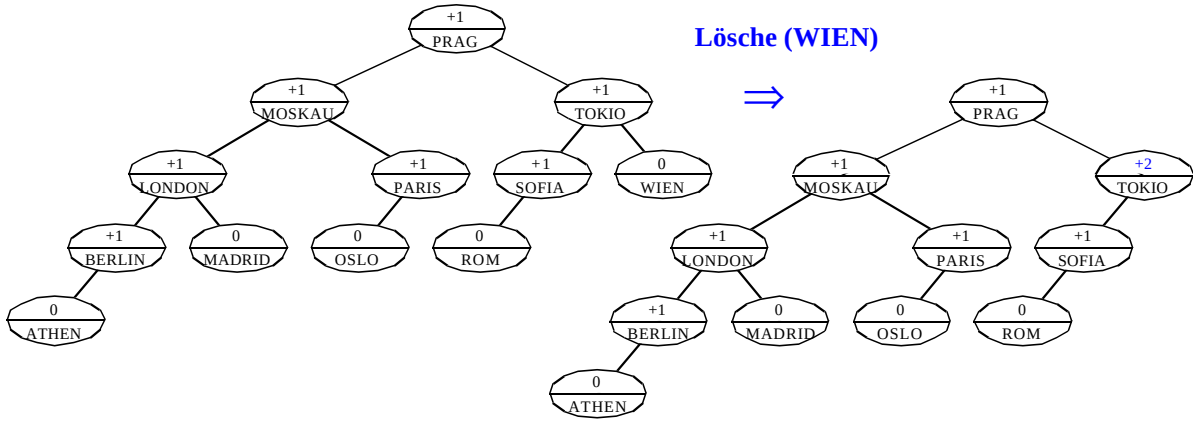
Löschen in AVL-Bäumen (6)

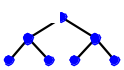
- Rebalancierungsschritte beim Löschen:

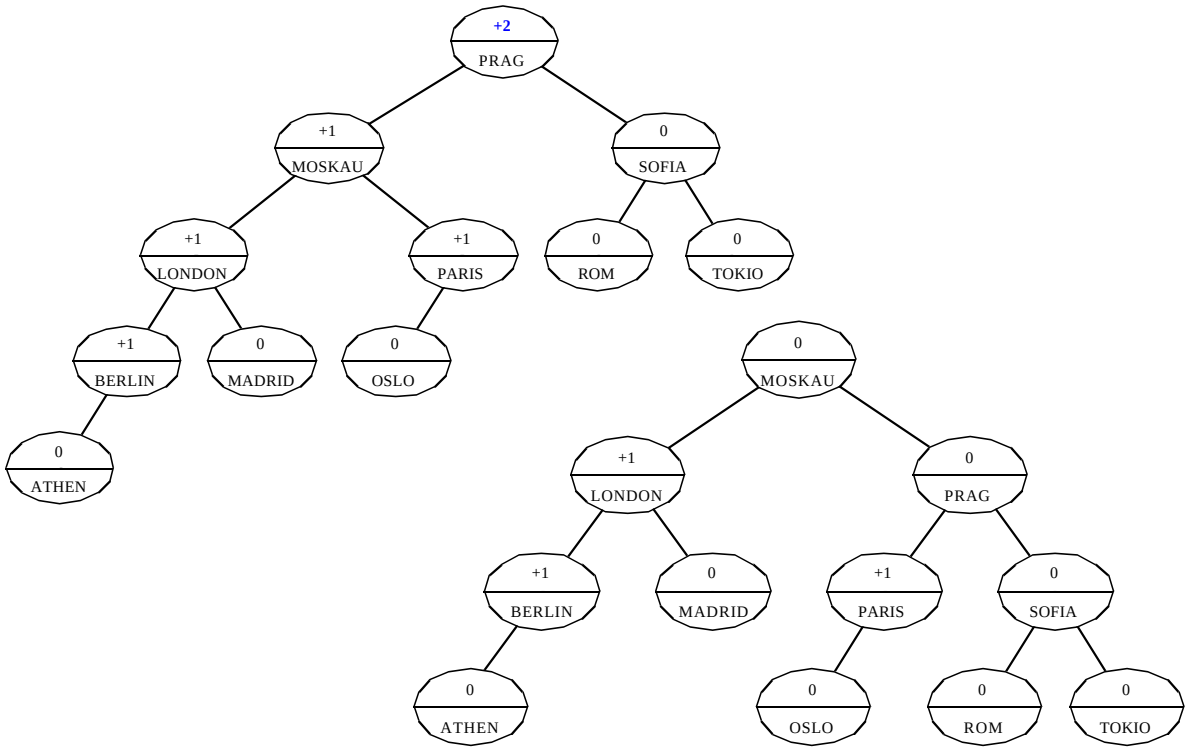
  1. Suche im Löschpfad nähesten Vater mit BF = ∓2 .

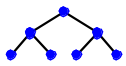
  2. Führe Rotation im gegenüberliegenden Unterbaum dieses Vaters aus.
  Im Gegensatz zum Einfügevorgang kann hier eine Rotation wiederum eine Rebalancierung auf dem Pfad zur Wurzel auslösen, da sie in gewissen Fällen auf eine Höhenerniedrigung des transformierten Unterbaums führt. Die Anzahl der Rebalancierungsschritte ist jedoch durch die Höhe h des Baums begrenzt

- Beispiel-Löschvorgang:









Höhe von AVL-Bäumen

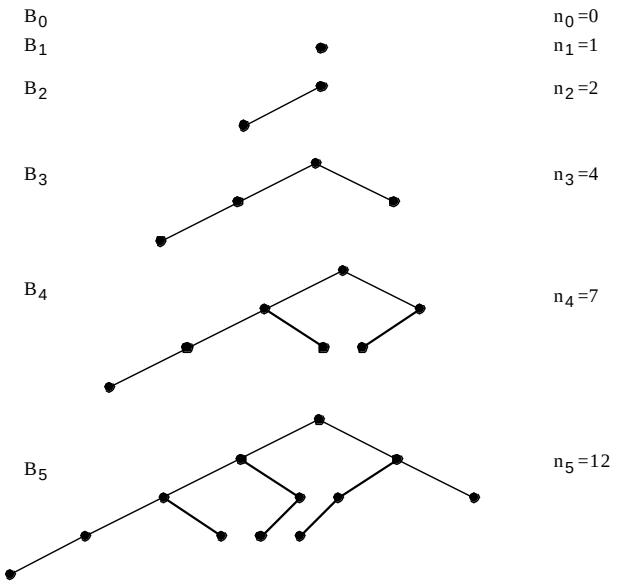
Balancierte Bäume wurden als Kompromiß zwischen ausgeglichenen und natürlichen Suchbäumen eingeführt, wobei logarithmischer Suchaufwand im schlechtesten Fall gefordert wurde

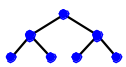
- Für die Höhe h_b eines AVL-Baumes mit n Knoten gilt:

$$\lfloor \log_2(n) \rfloor + 1 \leq h_b \leq 1{,}44 \cdot \log_2(n+1)$$

  - Die obere Schranke läßt sich durch sog. Fibonacci-Bäume, eine Unterklasse der AVL-Bäume, herleiten.

- Definition für **Fibonacci-Bäume** (Konstruktionsvorschrift)
  - Der leere Baum ist ein Fibonacci-Baum der Höhe 0.
  - Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1.
  - Sind B_{h-1} und B_{h-2} Fibonacci-Bäume der Höhe h-1 und h-2, so ist $B_h = \langle B_{h-1}, x, B_{h-2} \rangle$ ein Fibonacci-Baum der Höhe h
  - Keine anderen Bäume sind Fibonacci-Bäume





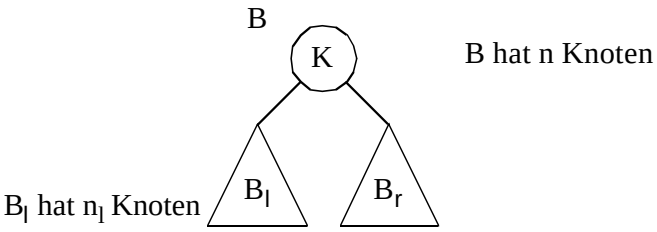
Gewichtsbalancierte Suchbäume

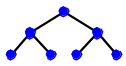
- Gewichtsbalancierte oder BB-Bäume (bounded balance)

  Zulässige Abweichung der Struktur vom ausgeglichenen Binärbaum wird als Differenz zwischen der Anzahl der Knoten im rechten und linken Unterbaum festgelegt

- Def.: Sei B ein binärer Suchbaum mit linkem Unterbaum B_l und sei n (n_l) die Anzahl der Knoten in B (B_l).
  - $\rho(B) = (n_l+1)/(n+1)$ heißt die Wurzelbalance von B.
  - Ein Baum B heißt gewichtsbalanciert (BB(α)) oder von beschränkter Balance α, wenn für jeden Unterbaum B' von B gilt:

$$\alpha <= \rho(B') <= 1 - \alpha$$



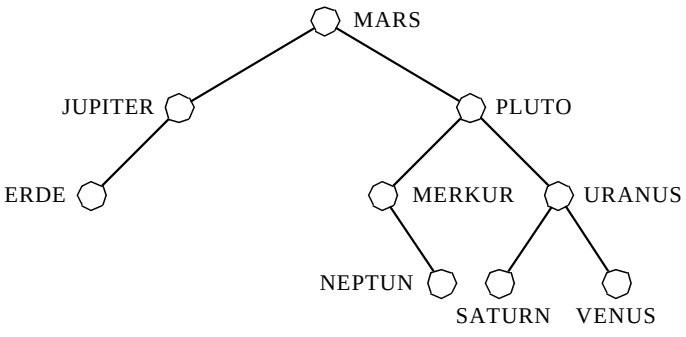


Gewichtsbalancierte Suchbäume (2)

- Parameter α als Freiheitsgrad im Baum
  - $\alpha = 1/2$: Balancierungskriterium akzeptiert nur vollständige Binärbäume
  - $\alpha < 1/2$: Strukturbeschränkung wird zunehmend gelockert

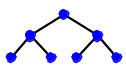
  Welche Auswirkungen hat die Lockerung des Balancierungskriteriums auf die Kosten ?

- Beispiel: Gewichtsbalancierter Baum in BB (α) für $\alpha = 3 / 10$



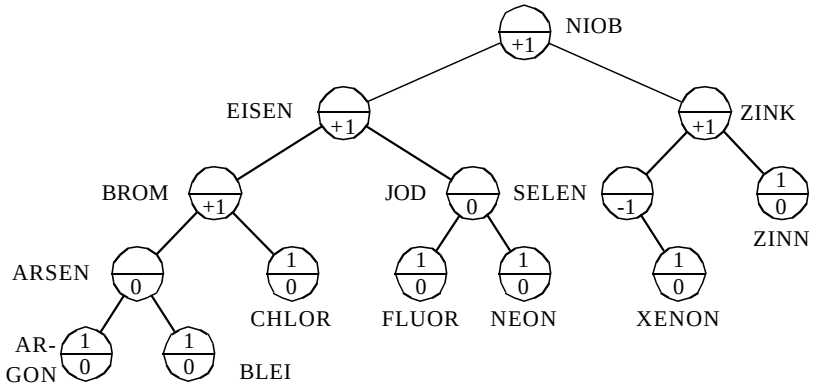
- Rebalancierung
  - ist gewährleistet durch eine Wahl von $\alpha <= 1 - \sqrt{2}/2$
  - Einsatz derselben Rotationstypen wie beim AVL-Baum

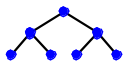
- Kosten für Suche und Aktualisierung: $O(\log_2 n)$



Positionssuche mit balancierten Bäumen

- Balancierte Suchbäume
  - sind linearen Listen in fast allen Grundoperationen überlegen
  - Lösung des Auswahlproblems bzw. Positionssuche (Suche nach k-tem Element der Sortierreihenfolge) kann jedoch noch verbessert werden
- Def.: Der Rang eines Knotens ist die um 1 erhöhte Anzahl der Knoten seines linken Unterbaums
  - Blattknoten haben Rang 1
- Verbesserung bei Positionssuche durch Aufnahme des Rangs in jedem Knoten
- Beispiel für AVL-Baum:

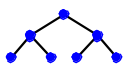




Positionssuche (2)

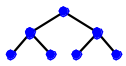
- Rangzahlen erlauben Bestimmung eines direkten Suchpfads im Baum für Positionssuche nach dem k-ten Element
  - Position p := k; beginne Suche am Wurzelknoten
  - Wenn Rang r eines Knotens = p gilt: Element gefunden
  - falls r > p, suche im linken UB des Knotens weiter
  - r < p => p := p - r und Fortsetzung der Suche im rechten UB
- Wartungsoperationen etwas komplexer

  Änderung im linken Unterbaum erfordert Ranganpassung aller betroffenen Väter bis zur Wurzel



Zusammenfassung

- Binäre Suchbäume
  - Einfügen / direkte Suche durch Baumdurchgang auf einem Pfad
  - Reihenfolgeabhängigkeit bezüglich Einfügeoperationen
  - sequentielle Suche / sortierte Ausgabe aller Elemente: Inorder-Baumtraversierung (O (n))
  - minimale Zugriffskosten der direkten Suche O (log n); mittlere Kosten Faktor 1,39 schlechter
- Balancierte Suchbäume zur Sicherstelleung günstiger Zugriffskosten
  - höhenbalancierte Bäume (z.B. k-balancierte Binärbäume)
  - gewichtsbalancierte Bäume (BB-Bäume)
- AVL-Baum: 1-balancierter Binärbaum
  - Mitführen eines Balancierungsfaktors B in Baumknoten (zulässige Werte: -1, 0, oder 1)
  - dynamische Rebalancierung bei Einfüge- und Löschoperationen (Fallunterscheidungen mit unterschiedlichen Rotationen)
  - Anzahl der Rebalancierungsschritte durch Höhe des Baumes begrenzt
  - maximale Höhe für Fibonacci-Bäume: 1,44 log (n)
- schnelle Positionssuche über Mitführen des Rangs von Knoten



Zusammenfassung: Listenoperationen auf verschiedenen Datenstrukturen

Operation	sequent. Liste	gekettete Liste	balancierter Baum mit Rang
Suche von K_i			
Suche nach k-tem Element			
Einfügen von K_i			
Löschen von K_i			
Löschen von k-tem Element			
sequentielle Suche