

Ordnung ist das halbe Leben

- wer Ordnung hält ist nur zu faul zum Suchen
- kreatives Chaos
- Unordnung ist die andere Hälfte
- Ordnung halten kostet mehr Zeit als suchen



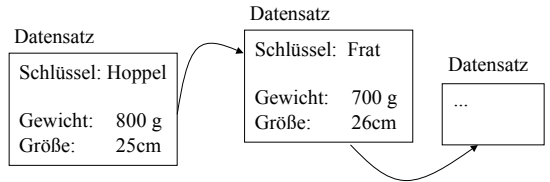
Dennoch:

Suchen ist eine der häufigsten und aufwendigsten Operationen in Datenverarbeitungssystemen.

- Beispiele:
- russische Übersetzung für englisches Wort transparency
 - englische Übersetzung für chinesisches Wort
 - Suchen der Adresse: Am Fasanengarten 3b
 - Suchen nach Person zu gefundenem Fingerabdruck
 - Suchen eines Ausgangs aus einem Labyrinth

Elektronische Datenverarbeitung EDV

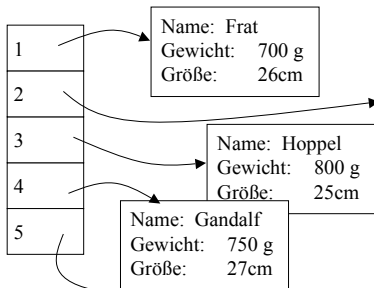
- Elektronische Daten bestehen aus Datensätzen
- Datensätze enthalten i.d.R.
 - „Nutzdaten“ = eigentliche Inhalte (oft als Felder)
 - „organisatorische Daten“ = Overhead
- Datensätze haben einen oder mehrere Schlüssel (meist eindeutige) zum Identifizieren des Datensatzes



Elektronische Datenverarbeitung EDV

Beliebtes Vorgehen: Verwenden von Stellvertreterliste:

Statt der Anordnung der eigentlichen Daten genügt es, die Stellvertreterliste zu organisieren.



Eine der meist genutzten Anwendungen

Internet-Suchmaschinen

Teilaufgaben:

- Dokumente analysieren und Teiltexpte (z.B. Wörter darin finden)
- Referenzen auf Dokumente sinnvoll organisiert abspeichern
- Anfragen wie: „in welchen Dokumenten kommt X vor“ beantworten

Spezialaufgaben:

- Dokumente finden, in denen X mit Tippfehlern drin steht
- Suchen nach als regulärer Ausdruck gegebenem X (z.B. ?s . [0-9] *)

Sachen Wiederfinden

Internet-Suchmaschine:

Dieser Algorithmus wäre schön blöd:

```
input Suchbegriff;
for (alle Webseiten der Welt)
  if (Suchbegriff in Webseite)
    füge Webseite zur Liste hinzu;
```

kaum besser wäre

```
input Suchbegriff;
für (jedes Wort im Wörterbuch)
  if (Suchbegriff = Wort) gib Webseitenliste aus
```

Sachen Einräumen und Wiederfinden

Übliche Operationen in der EDV:

- Datensatz erstellen / erfassen / eingeben
- Datensatz anzeigen / ausgeben
- Datensatz zur Datei hinzufügen
- Datensatz aus der Datei löschen
- Datensatz für gegebenen Schlüssel suchen / finden

} I/O-Maske

Definitionsbereich der Schlüsselmenge

Beispielproblem 1: Wörterbuch / Vokabelbuch (Dictionary)

- gegeben: Wort (String) in einer Quellsprache
- gesucht: entsprechendes Wort (oder mehrere) in der Zielsprache
- binäre Suche: $O(\log n)$, d.h. bei 1 000 000 Wörtern ca. 20 „Einstiche“

Beispielproblem 2: Indexierung von Wörtern (String Integer)

- Integers sparen (i.d.R.) Speicherplatz
- Integers haben immer konstante Länge (einfache Speicherallokation)
- mit Integers kann man arithmetische Operationen durchführen
- Integers eignen sich als Array- und Tabellenindizes
- sinnvoller Einsatz z.B. bei Bezeichnungen in Programmiersprachen

Hashing / Streuen

Beliebtes Verfahren zur Abbildung von komplexen Schlüssel in einfache (z.B. Integers) heißt:

deutsch: hashing
streuen oder haschen

Aus Langenscheidt Wörterbuch Englisch:

hash¹ [hæʃ] 1. gehacktes Fleisch *n*;
Am. F Essen *n*, Fraß *m*; *fig.* Misch-
masch *m*; *make a ~ of F et.* ver-
pfuschen; *settle s.o.'s ~ F* es j-m
besorgen; 2. (zer)hacken.
hash² F [-] Hasch(isch) *n*.

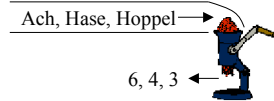
Hashing / Streuen

Definiere Abbildung von
Schlüsselmenge nach Indexmenge

$f(\text{Schlüssel}) = \text{Index}$

z.B. für Schlüsseltyp String

$f(\text{Schlüssel}) = \text{Schlüssel.length}()$



Hash-Tabelle

0	
1	
2	
3	Ach
4	Hase
5	
6	Hoppel
7	
8	Computer
9	

Hashing / Streuen

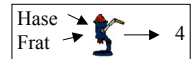
Wenn Hash-Tabelle m Zeilen hat und $|\text{def}(f)| = n$,
ist der **Belegungsfaktor** n/m

Keine Probleme gibt es, wenn die Hash-Funktion **injektiv** ist
 $\forall \text{Schlüssel-1} \neq \text{Schlüssel-2}: f(\text{Schlüssel-1}) \neq f(\text{Schlüssel-2})$

Der Fall $f(\text{Schlüssel-1}) = f(\text{Schlüssel-2})$ wird **Kollision** genannt.

Kollisionsfreiheit erfordert meist:

- sehr große Hash-Tabelle
- clevere (da notwendigerweise injektive) Hash-Funktion



In der Praxis ist Kollisionsfreiheit in den seltensten Fällen machbar

Hashing: Kollisionsvermeidung

Was tun, damit $f(\text{Schlüssel-1}) \neq f(\text{Schlüssel-2})$?

- Hash-Tabelle mindestens so groß wie zu „hashende“ Schlüsselmenge
- Hash-Funktion f geschickt wählen (aber nicht zu aufwendig)

z.B. Hash-Tabelle für 2-Zeichen-Variablenbezeichner (Basic)

Schlüsselmenge $S = \{ A,B,\dots,Z,AA,AB,\dots,AZ,A0, \dots, A9,BA,\dots,Z9 \}$
 $|S| = 26 + 26 \cdot 36 = 962$

$f(A)=0, f(B)=1, \dots, f(Z)=35, f(AA)=36, f(AB)=37, \dots,$
 $f(AZ)=71, f(A0)=72, \dots, f(A9)=81, f(BA)=82, \dots, f(Z9) = 962$

ist kollisionsfrei, füllt die Tabelle zu 100% aus (Belegungsfaktor 1.0)

Hashing: Kollisionsauflösung

In der Praxis: Kollision nicht auszuschließen,
aber Kollisionswahrscheinlichkeit minimieren

$f(\text{Schlüssel})$ zufällig wählen? z.B.: `function f (Schlüssel)
seed := Schlüssel;
return random(1..n)`

Tabelle habe $m = 365$ Einträge
Schlüsselmenge habe 23 Elemente
(Geburtstagsproblem von *von Mises*)

$p(\text{irgend eine Kollision}) > 0.5$

Hashing: Kollisionsauflösung

Was tun, wenn trotzdem $f(\text{Schlüssel-1}) = f(\text{Schlüssel-2})$?

1. Idee: wenn Tabellenzeile $f(\text{Schlüssel})$ belegt, dann verwende $f'(\text{Schlüssel}) = f(\text{Schlüssel}) + 1$, bei erneuter Kollision (Sekundärkollision), $f'' = f' + 1$ usw.

D.h.: nimm den nächsten freien Platz in der Tabelle ab $f(\text{Schlüssel})$ ggf. nochmal zyklisch von vorne anfangend.

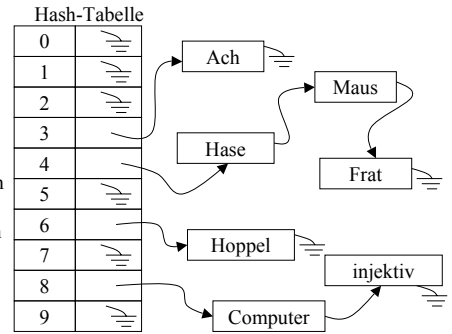
Verfahren heißt lineares Sondieren (*linear probing*).

Geschicktere Wahl von f' (besser als $f+1$) kann dafür sorgen, daß sich Schlüssel mit gleicher Hash-Funktion nicht lokal ballen.

Hashing: Kollisionsauflösung

2. Idee:

Kollisionsauflösung durch Verkettung, hat außer der Hash-Tabelle noch einen besonderen Speicherbereich, in dem die Daten selbst als verzeigerte Listen stehen.



Hashing: Kollisionsauflösung

Methode	Vorteile	Nachteile
Verkettung	einfache Hash-Funktion geeignet für „große“ Basistypen oder für unbekannte Schlüssel-mengen (bel. große)	zusätzlicher Speicher dynamisch alloziert zusätzlicher ADT Liste incl. Operationen
Sondieren	konstanter Speicherplatz geeignet für „kleine“ Basistypen, bekannte Schlüssel-mengen	kompliziertere mehr-stufige Hash-Funktion nur ein ADT => einfach Entfernen sehr teuer

Hashing: Aufwandsschätzung

für die „Verwaltung“ einer Hash-Tabelle für n Elemente:

Berechnung von $f(\text{Schlüssel})$ sollte in $O(1)$ also unabh. von n gehen.

Einsortieren in Tabelle ohne Kollision: $O(1)$
mit Kollision: $O(n \cdot p)$

insgesamt also $(1-p)O(1) + p \cdot O(n \cdot p)$

wobei $0 \leq p \leq 1$ die Wahrscheinlichkeit für eine Einzelfall-Kollision ist.

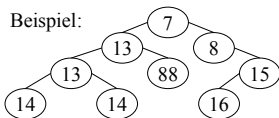
also ist der Aufwand zwischen $O(1)$ für $p=0.0$ und $O(n)$ für $p=1.0$.

Der Heap (Die Halde)

Definition:

- ein Heap ist ein abstrakter Datentyp
- ein Heap ist ein Binärbaum
- jedem Knoten ist ein Datum (Schlüssel) zugeordnet
- für jeden Knoten eines Heaps gilt:
sein Schlüssel ist in der Sortierung vor allen Schlüsseln aller Nachfolgerknoten

Beispiel:



Der Wurzelknoten hat den „kleinsten“ Wert.

Machbar auch: Sortierung von groß nach klein.

Heaps: Operationen

Der ADT Heap ist definiert über die Schnittstelle:

`void insert(x)` fügt Element x in den Heap ein
`Object remove(void)` entfernt kleinstes Element des Heaps

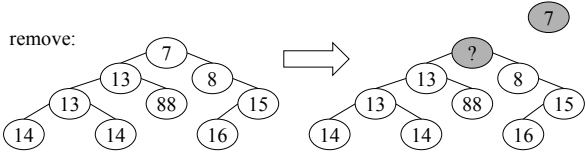
Jede Operation garantiert, daß der ADT danach immer noch Heap ist.

Heaps sind geeignet um priorisierte Warteschlange zu implementieren. Zum Beispiel Prozesse eines Multitasking Systems:

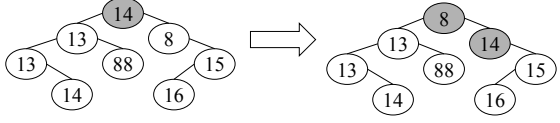
`insert(x)` neuen Prozeß in die Schlange einreihen
`remove()` holt den wichtigsten Prozeß aus der Schlange

Heaps: Operationen

remove:



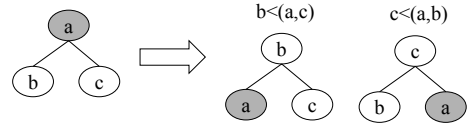
ersetze Wurzel durch ein Blatt, z.B. 14



Heaps: Operationen

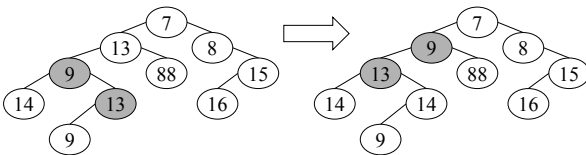
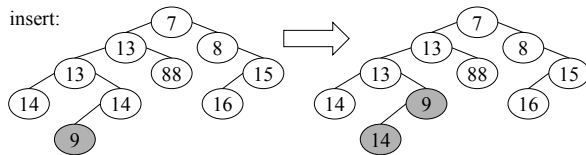
Algorithmus für remove-Operation:

- ersetze Wurzel durch einen Blattknoten
- aktueller Knoten = Wurzel
- solange aktueller Knoten größer als einer seiner Nachfolgeknoten
 - vertausche ihn mit dem kleineren seiner Nachfolger
 - setze aktuellen Knoten = was der kleinere Nachfolger war



Heaps: Operationen

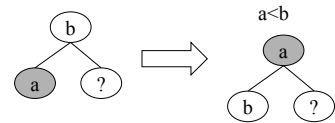
insert:



Heaps: Operationen

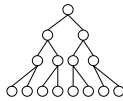
Algorithmus für insert-Operation:

- füge neuen Blattknoten mit neuem Schlüssel hinzu
- aktueller Knoten = neues Blatt
- solange aktueller Knoten kleiner als sein Vorgängerknoten
 - vertausche ihn mit dem Vorgänger
 - setze aktuellen Knoten auf was Vorgänger war



Heaps: Aufwandsschätzung

Wenn der Binärbaum in etwa ausbalanciert ist (bei n Knoten hat der Baum eine Tiefe von log n)



remove: ein Element von oben durch maximal alle Ebenen nach unten durchpropagieren => O(log n)

insert: ein Blatt von unten durch maximal alle Ebenen nach oben durchpropagieren => O(log n)

Allerdings: keine Suchoperation vorgesehen, bräuchte mindestens einen Aufwand von O(n)

Dynamisches Programmieren

Motivation:

Auffinden von Textstücken

- z.B. in Dokumenten
- ggf. auch nicht 100%-ig korrekte Stellen

gesucht: Bundeskanzler
... im Bundestag sagte der Bundeskanzler, er ...

...sagte der Bundeskanzler, er (fuzzy match)

Minimale Editierdistanz

- getippt: pribic gemeint private oder public ?

Fehlermessung bei ganzen Sätzen

- z.B. Spracherkennung: gesagt „Heute ist schönes Wetter.“ erkannt „Heute schönes Wetter ach.“ Frage: wie viele Fehler?

Mustererkennung

- Bilder, Tonaufnahmen sind Folgen von Meßwerten
Mustererkennung = Vergleich mit gespeicherten Meßwertfolgen

Vergleich von Symbolsequenzen

Aufgabe: finde die minimale Editierdistanz zwischen zwei Symbolfolgen $A = a_1, a_2, \dots, a_n$ und $B = b_1, b_2, \dots, b_m$

Editierdistanz = Anzahl der Editierschritte, um aus A zu B zu machen

- z.B. $A = \text{prbic}$ $B = \text{public}$
- Schritt: lasse das r in A weg $\text{prbic} \Rightarrow \text{pibic}$
 - Schritt: ersetze erstes i durch u $\text{pibic} \Rightarrow \text{public}$
 - Schritt: füge ein l nach dem b ein $\text{public} \Rightarrow \text{public}$

In weniger als 3 Schritten geht es nicht, in mehr als 3 aber wohl.

Es gibt drei verschiedene möglich Schritte:

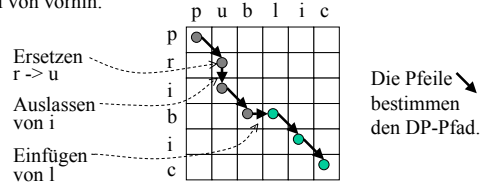
Weglassen Ersetzen Einfügen

Vergleich von Symbolsequenzen

Die DP-Matrix

Dynamisches Programmieren (DP) wird oft als Matrix dargestellt.

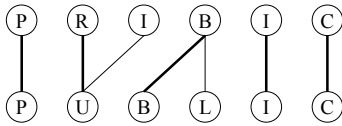
Beispiel von vorher:



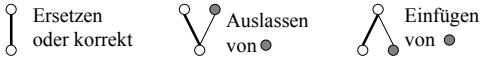
Die Pfeile bestimmen den DP-Pfad.

Vergleich von Symbolsequenzen

Andere Darstellung (z.B. als bipartites Matching):



wobei

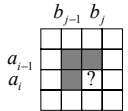


Vergleich von Symbolsequenzen

Induktiver Algorithmus, der die minimale Editierdistanz findet:

gegeben: $A = a_1, a_2, \dots, a_n$ $B = b_1, b_2, \dots, b_m$

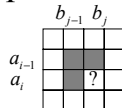
Annahme: Wir wissen, was die minimale Editierdistanz zwischen a_1, a_2, \dots, a_{i-1} und b_1, b_2, \dots, b_{j-1} sowie zwischen a_1, a_2, \dots, a_{i-1} und b_1, b_2, \dots, b_j sowie zwischen a_1, a_2, \dots, a_i und b_1, b_2, \dots, b_{j-1} ist



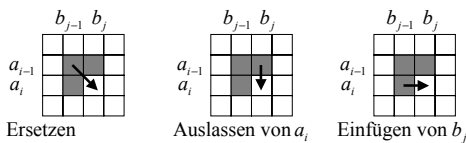
Wie berechnet sich die minimale Editierdistanz zwischen a_1, a_2, \dots, a_i und b_1, b_2, \dots, b_j ?

Vergleich von Symbolsequenzen

Bezeichne $C(i, j)$ die minimale Editierdistanz zwischen a_1, a_2, \dots, a_i und b_1, b_2, \dots, b_j

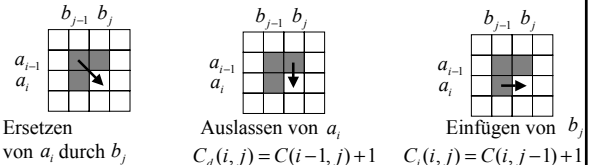


Ausgangslage: Bekannt sind $C(i-1, j-1)$, $C(i-1, j)$ und $C(i, j-1)$, dann sind mögliche Erweiterungen der DP-Pfade zur Matrixzelle (i, j) :



Vergleich von Symbolsequenzen

Aktualisieren der mitgeführten (akkumulierten) Editierdistanz:



$C_d(i, j) = C(i-1, j) + 1$ $C_i(i, j) = C(i, j-1) + 1$

$C_s(i, j) = C(i-1, j-1)$

$+ \begin{cases} 0 & \text{falls } a_i = b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases}$

also $C(i, j) = \min \begin{cases} C_s(i, j) \\ C_d(i, j) \\ C_i(i, j) \end{cases}$

Finden der minimalen Editierdistanz

	H	A	S	E	
M	M≠H: 1 Σ=1	M≠A: 1 Σ=2	M≠S: 1 Σ=3	M≠E: 1 Σ=4	H A S E I I I I
A	A≠H: 1 Σ=2	A=A: 0 Σ=1	A≠S: 1 Σ=2	A≠E: 1 Σ=3	
U	U≠H: 1 Σ=3	U≠A: 1 Σ=2	U≠S: 1 Σ=2	U≠E: 1 Σ=3	H A S E I I I I M A U S
S	S≠H: 1 Σ=4	S≠A: 1 Σ=3	S=S: 0 Σ=2	S≠E: 1 Σ=3	

Vergleich von Symbolsequenzen

Andere Distanzmaße:
manchmal werden verschiedene Fehlerarten unterschiedlich schwer gewichtet, z.B. Vertauschung von X und U ist schlimmer (oder unwahrscheinlicher) als I und L.

Allgemein also: $C_d(i, j) = C(i-1, j) + c_d(a_i)$ Kosten für Auslassen von a_i

$C_i(i, j) = C(i, j-1) + c_i(b_j)$ Kosten für Einfügen von b_j

$C_i(i, j) = C(i-1, j-1) + c_s(a_i, b_j)$ Kosten für Ersetzen von a_i durch b_j

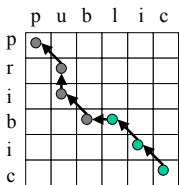
Vergleich von Symbolsequenzen

Wie findet man den DP-Pfad? DP-Schritt: $C(i, j) = \min \begin{cases} C_s(i, j) \\ C_d(i, j) \\ C_i(i, j) \end{cases}$

Merke zusätzlich: $M(i, j) = \begin{cases} (i-1, j-1) & \text{falls } C(i, j) = C_s(i, j) \\ (i-1, j) & \text{falls } C(i, j) = C_d(i, j) \\ (i, j-1) & \text{falls } C(i, j) = C_i(i, j) \end{cases}$

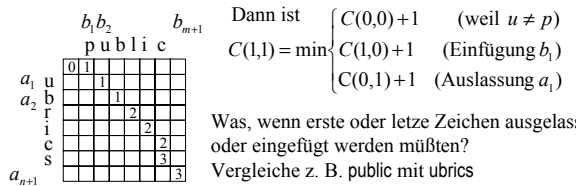
$M(i, j)$ zeigt auf die beste Vorgängerzelle.

Pfad: $(n, m), M(n, m), M(M(n, m)), \dots, M(M(\dots(n, m) \dots))$



Vergleich von Symbolsequenzen

Randbedingungen: z.B. Definiere $C(i, 0) = C(0, j) = \infty \forall i, j \neq 0$ und $C(0, 0) = 0$



$C(i, j)$ entlang des Pfades der minimalen Editierdistanz Dann: Definiere $C(i, 0) = i, C(0, j) = j$

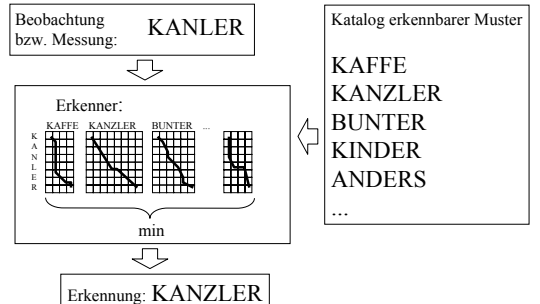
Vergleich von Symbolsequenzen

Matrix der lokalen (nicht kumulativen) Distanzen:

		b_1	b_2		b_{m+1}	
		p	u	b	i	c
a_1	u	0	1	1	1	1
a_2	b	1	1	0	1	1
	r	1	1	1	0	1
	i	1	1	1	1	0
	c	1	1	1	1	1
	s	1	1	1	1	1
a_{n+1}		0	1	1	1	0

Ziel des DP: Finde einen Pfad, der möglichst wenige senkrechte / waagerechte Schritte macht und möglichst wenige lokale Distanzen aufsummiert.

Mustererkennung mit dynamischem Programmieren



DP mit Wahrscheinlichkeiten

Motivation: OCR liest Zeitung und erkennt PANZIER
War das eher PANZER oder KANZLER?

Editierdistanz zu PANZER ist 1,
Editierdistanz zu KANZLER ist 2,

Was aber, wenn es wahrscheinlicher ist, daß K mit P
und L mit I verwechselt wird, als daß ein I eingefügt wird?

Möglicher Ansatz: setze $c_d(I) > c_s(K, P) + c_s(L, I)$
u. a. Kostendefinitionen

Ordnen und Suchen: Binärbäume

Heaps erlauben Einfügen und Entfernen (ausgezeichnetes Element)
in jeweils $O(\log n)$ Zeit bei n Elementen im balancierten Heap.

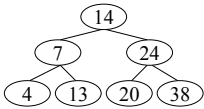
Aber Suche würde im worst-case $O(n)$ dauern.
Und Balance ist nicht garantiert, so daß auch insert und remove
jeweils im worst-case $O(n)$ benötigen.

Wünschenswert wäre ein ADT, der dafür sorgt,

- daß der Baum ausbalanciert ist
- schnelle Suche (in $O(\log n)$) möglich ist
- schnelles Einfügen (in $O(\log n)$) möglich ist
- schnelles Löschen beliebiger Elemente (in $O(\log n)$) möglich ist

Ordnen und Suchen: Binärbäume

4 7 13 14 20 24 38



```
class Binaerbaum
{ class Knoten
  { int Schlüssel;
    Knoten links;
    Knoten rechts;
  }
  Knoten Wurzel;
}
```

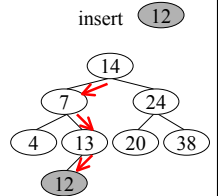
```
Knoten suche(Knoten k, int x)
{ if (k==null) return k;
  if (k.Schlüssel==x) return k;
  if (k.Schlüssel<x) return suche(k.rechts,x);
  if (k.Schlüssel>x) return suche(k.links,x);
}
```

Ordnen und Suchen: Binärbäume

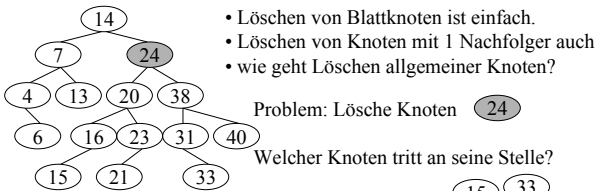
```
void einfuege (Knoten k, int x)
```

```
{
  if (k.Schlüssel < x)
    if (k.rechts==null)
      {k.rechts = new Knoten(x);
       return;}
    else einfuege(k.rechts,x);

  if (k.Schlüssel > x)
    if (k.links==null)
      {k.links = new Knoten(x);
       return;}
    else einfuege(k.links,x);
}
```



Ordnen und Suchen: Binärbäume



- Löschen von Blattknoten ist einfach.
- Löschen von Knoten mit 1 Nachfolger auch
- wie geht Löschen allgemeiner Knoten?

Problem: Lösche Knoten 24

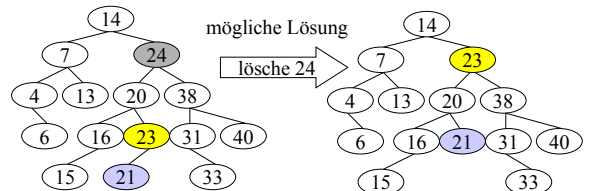
Welcher Knoten tritt an seine Stelle?

Keines der Blätter 6, 15, 33, 21, 40 paßt

Bedingungen für Ersatz von Knoten 24 :

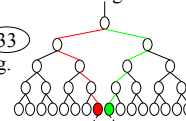
- $x \geq 14$ (da rechter Nachfolger von 14)
- $20 \leq x \leq 38$ (da 20 linker und 38 rechter Nachfolger wird)
- $23 \leq x \leq 31$ (da 23 größter Nachfolger von 20 und 31 kleinster Nachfolger von 38 ist)

Ordnen und Suchen: Binärbäume



auch Ersetzen durch die 31 und Aufrücken der 33 wäre eine Lösung.

Allgemein:



ersetze zu löschenden Knoten durch größten Knoten im linken Teilbaum oder kleinsten Knoten im rechten Teilbaum (der hat max. 1 Nachfolger)

Ordnen und Suchen: Binärbäume

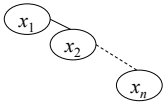
Suche in Binärbäumen geht also in $O(\text{Tiefe}) = O(\log n)$ für balancierte Bäume

Einfügen in Binärbäume geht in $O(\text{Tiefe}) = O(\log n)$ für balancierte Bäume

Löschen aus Binärbäumen geht in $O(\text{Tiefe}) = O(\log n)$ für balancierte Bäume

Einziges Problem noch: wie kriegen wir die Bäume balanciert?

Z.B. Einfügen einer sortierten Folge $x_1, x_2, x_3, \dots, x_n$ liefert:

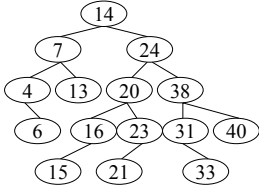


Ordnen und Suchen: AVL-Bäume

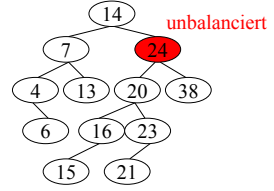
AVL steht für Adel'son, Vel'skii, Landis

AVL-Bäume haben die Eigenschaft, daß der Höhenunterschied von linken und rechten Teilbäumen maximal 1 ist.

korrekter AVL-Baum



kein AVL-Baum



Ordnen und Suchen: AVL-Bäume

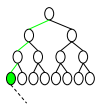
AVL-Bäume sind Binäre Suchbäume: Suchen, Einfügen, Löschen im Prinzip wie bei unbalancierten Bäumen, zusätzlich: **Ausbalancieren**

Balance prüfen in einem Baum mit n Knoten benötigt $O(n)$ Zeit.

Daher:

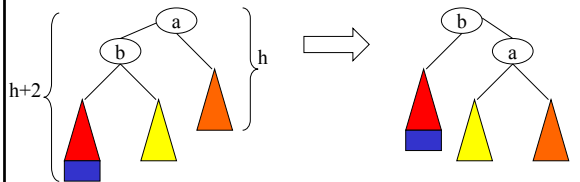
- jeder Knoten merkt sich seinen **Balanzfaktor** = Höhe(linker Teilbaum) - Höhe(rechter Teilbaum)

- bei jeder Änderung genügt es, die Balancefaktoren auf einem Baumast (in $O(\log n)$) zu aktualisieren



Ordnen und Suchen: AVL-Bäume

Ausbalancieren geht durch „Rotation“:

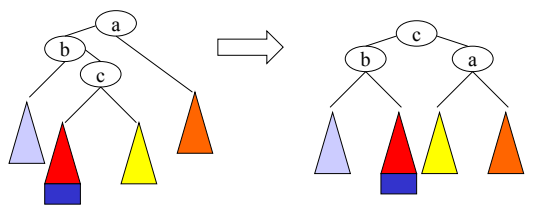


Durch **Einfügen** ist der **rote** Ast um 2 höher als der **orange**farbige Ast geworden

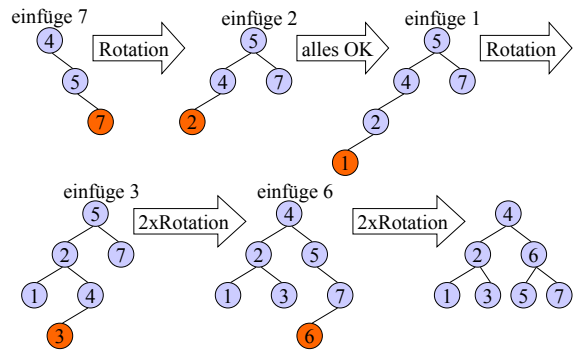
Umhängen der Teilbäume geht in $O(1)$ (der Haken von Δ wird an a gehängt und b zur neuen Wurzel)

Ordnen und Suchen: AVL-Bäume

Ausbalancieren benötigt manchmal auch „Doppelrotation“:



AVL-Bäume: Ein Beispiel



Sortieren

- ist die Grundlage zum Wiederfinden von Datensätzen
- besonders kritisch für sehr große Dateien
- einige Eigenschaften von Sortieralgorithmen
 - Laufzeit (Zeitkomplexität)
 - *in-place* oder nicht *in-place* (Speicherkomplexität)
 - Stabilität (Reihenfolgeerhalten für gleiche Schlüssel)

Sortieren: Heapsort

bereits kennengelernt: Heap, den können wir zum Sortieren nutzen:

gegeben float a[] und int n, und ADT-Klasse Heap, Sortieralgorithmus:

```
heap = new Heap(n);
for (int i=0; i<n; i++) heap.insert(a[i]);
for (int i=0; i<n; i++) a[i] = heap.remove();
```

Aufwandsschätzung:

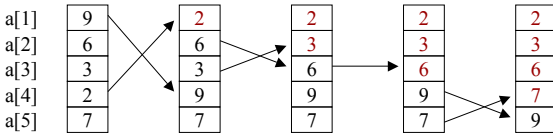
Platz: $O(n)$ n floats in a[] und n floats in heap,
(Es ist möglich, den Heap in a[] aufzubauen → Übg.)

Zeit: $2n \cdot O(\log n) = O(n \log n)$

Sortieren mit Feldern (Arrays)

Selection-Sort (Sortieren durch Auswahl):

```
for (i=1..n-1)
{
    m=i; for (j=i+1..n) if (a[j]<a[m]) m=j;
    vertausche a[i] mit a[m]
}
```

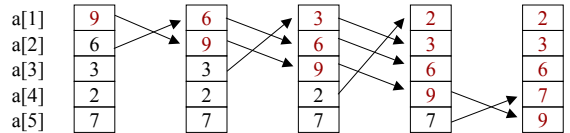


Aufwand: Offensichtlich $O(n^2)$

Sortieren mit Feldern (Arrays)

Insertion-Sort (Sortieren durch Einordnen):

```
for (i=2..n)
{
    for (j=1..i-1) if (a[j]>a[i]) break;
    a[i] vor a[j] ein
}
```

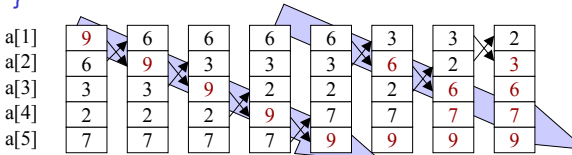


Aufwand: Offensichtlich $O(n^2)$

Sortieren mit Feldern (Arrays)

Bubble-Sort (Sortieren durch Austauschen):

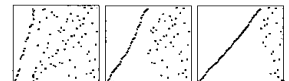
```
for (i=n..2)
{
    for (j=1..i)
        if (a[j]>a[j+1])
            vertausche a[j] mit a[j+1];
}
```



Aufwand: Offensichtlich $O(n^2)$. Im best-case $O(n)$ ⇒ geeignet für fast sortierte Arrays

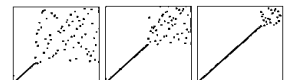
Sortieren mit Feldern (Arrays)

Graphischer Vergleich verschiedener Sortierverfahren:



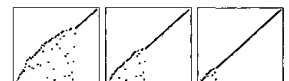
Einsetzungssortierung

Vorteil von Sortieren durch Einordnen (Einsetzen):



Selektionssortierung

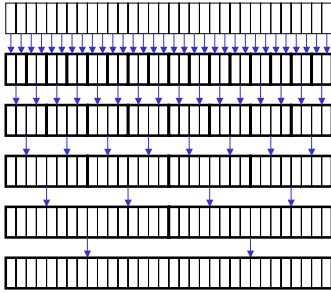
Verschieben von Teil-Arrays nicht nötig, wenn verzeigerte Listen sortiert werden (dann nur Zeiger verbiegen).



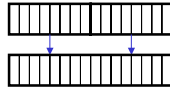
Blaasensortierung

Sortieren mit Feldern (Arrays)

Merge-Sort (Sortieren durch Mischen):



sortiert sortiert



zwei sortierte Teilfelder werden im Reißverschlussverfahren zu einem sortierten Feld gemischt.

Geht nicht in-place.

Aufwand: $O(n \log n)$

Sortieren mit Feldern (Arrays)

Quicksort: Algorithmusschema:

- teile Feld in zwei Teile A, B, so daß alle Elemente in A < alle in B
- rufe Quicksort jeweils für A und für B auf
- zur Aufteilung in A und B:
 - wähle beliebiges Element des Feldes als „**Pivot-Element**“
 - ordne alle $a[i] < \text{Pivot}$ in A und alle $a[i] \geq \text{Pivot}$ in B (geht in $O(n)$)

Im günstigsten Fall ist $|A| = |B|$ (exakt halbiert): $T(n) = 2 \cdot T(n/2) + O(n)$
 $= O(n \log n)$

Im ungünstigsten Fall ist $|A|=1$ oder $|B|=1$: $T(n) = T(n-1) + O(n)$
 $= O(n^2)$