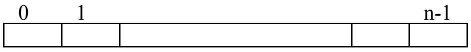
Algorithmen und Datenstrukturen

Inhalt

1. Grundlagen

1.1. Maschinenmodell RAM

Speicher:



N Speicherzellen

Zugriff auf i-te Zelle kostet ein Rechenschritt (Elementare Operation)

Weitere elementare Operationen (d.h. 1 Schritt): Wertzuweisung, Arithmetik, Vergleiche

1.2. Pseudo-Code (einfache Algorithmen)

Statt einer konkreten Programmiersprache (Java, C++ …) verwenden wir eine abstrakte Sprache

Pseudo-Code [keine formale Definition]

Variabel: ohne Deklaration

Felder: A[l..r] A[0..n-1] B[1..n]

Statements und Kontrollstrukturen:

Wertzuweisung: i ← 17 j ←i

Arithmetik: +, -, *, /, +=, -=, --, ++

Vergleiche: =, ≠, <, >, ≤, ≥

Feldzugriff: A[i]

Kontrollstrukturen:

```
WHILE (i>0) DO
        [ ] Rumpf
OD
IF (n=100) THEN
        [ ]
(ELSE
        [ ] )
FI
```

```
FOR i=0 to 100 Do
        []
OD
FOR i=100 DOWNTO 1
        [ ]
OD
REPEAT:
        [ ]
UNTIL n=0
```

Unterprogramme: (Funktionen oder Prozeduren)

Alg(I,j,k) [ ]

Beispiel: Lineare Suche

Problem: Feld A[1..n] enthält die Zahlen {1,2,..n} in beliebiger Reihenfolge

Aufgabe: Für x={1..n} bestimme mit A[i]=x

d.h. Variable des Suchproblems ist immer erfolgreich

```
LinearSuch(x)
i← 1;
WHILE A[i]≠x DO
i ←i+1
OD
```

Bei allgemeiner Suche (evtl. x≠A[i]): WHILE (i≤n) ^ (A[i]≠x) DO …

Analyse der Laufzeit:

1. Laufzeit im schlechtesten Fall (worst Case)
Obere Schranke für die Zahl der Schleifendurchläufe ist n, wenn x=A[n] Zahl der Schritte: 4n+1 → Lineare Laufzeit
2. Laufzeit im mittleren Fall (erwartete Laufzeit)
Annahmen über die Eingabe erforderlich: jede Eingabe (jede n! Permutation) ist gleich wahrscheinlich
Für jedes i zwischen 1..n (1>i>n) gilt grob: (A[i]=x)=1/n
Mittlere Laufzeit: Erwartungswert der Anzahl der Schritte unter der Annahme T(n)=4n+1 (worst case)

Mittlere Laufzeit $\overline{T}(n) = \sum_{i=1}^{n}(4i+1)\cdot\frac{1}{n}$

Wahrscheinlichkeit x=i

Summe aller Möglichkeiten Kosten für x=A[i]

$$\overline{T}(n) = \frac{1}{n}\sum_{i=1}^{n}(5i) = \frac{5}{n}\sum_{i=1}^{n} i$$

$$\overline{T}(n) \approx \frac{5}{n}\left(\frac{1}{2}n(n+1)\right) \approx \frac{5}{2}(n+1) = \frac{5}{2}n + \frac{5}{2} \le 3n$$

Arithmetische Reihe
Allgemeine Suche: Beliebige Zahlen für A Suche nach beliebigen x
Trick Erzwinge den Erfolgsfall A[n+1]←x (Sentinel)

1.3. Asymptotische Laufzeit (O-Notation)

Idee: Teile alle Laufzeitfunktionen nach ihrem Asymptotischen Verhalten n → ∞ in Klassen ein
Definition: (O-Notation)

1. Obere Schranke:

Bei f: IN → IR_0^+

$$O(f) = [g : IN \to IR_0^+] c > 0 n_0 \in IN \quad \forall n \ge n_0 = c \cdot f(n)$$

2. Untere Schranke

$$\Omega(f) = (g : IN \to IR_0^+ \mid \exists c \ge 0, n_0 \in IN \quad \forall n \ge n_0 \quad g(n) \ge c \cdot f(x)]$$

3. Scharfe Schranke

$$\Theta(f) = [g : IN \to IR \mid c_1 \ge 0, c_2 \ge 0, n_0 \in IN \quad \forall n \ge n_0 \quad c_1 \cdot f(n) < g(n) < c_2 \cdot f(n)]$$

Schreibweise: Statt g $\in O(f)$ schreibt man g=O(f)

Beispiel: O-Notation

a) f(n) =3n+5
O(n), da f(n)_<4n für n≥5
c n₀
analog f(n) = Ω(n)
und f(n) = Θ(n)

b) f(n) = 2n² + 3n + 5
≤ 4n für n≥5
≤ 2n² + 4n für n≥5
≤ 2n² +n² für n≥4
≤ 3n² für n≥5

$=O(n^2)$ mit $c=3$ $n_0=5$
Allgemein für Polynome
$F(n)=a_k n^k+a_{k-1}n^{k-1}+\ldots+a_0 k^0$
Es gilt $f(n)=\Theta(n^k)$

O-Notation
Laufzeit $T(n)=c$ für eine konstante c
Bsp. Rumpf einer Schleife
WHILE (i<n) DO
C Elementare Operationen
Nicht abhängig von n

$T(n)\leq C$ $T(n)=O(1)$
$T(n)\geq C$ $T(n)=\Omega(1)$

1.4. Rekursive Algorithmen und Laufzeitgleichungen

Wende folgende Algorithmus rekursiv auf ein Teilproblem an (bis zur Verankerung)
Hier: „Die Divide and Conquer" - Methode (➔ Teile und Beherrsche)
3 Schritte:
1) Teile: teile das Problem (der Größe n) in k Teilprobleme (eventtuell der Größe n/k) der gleichen Art
2) Beherrsche: Löse die k-Teilprobleme – rekursiv d.h. mit diesem Algorithmen Falls das Teilproblem klein genug (konstant) löse es direkt (ohne Rekursion)
3) Zusammensetzung oder Mischen: Konstruiere die Lösung für das Gesamtproblem (Größe n) aus den k-Teillösungen aus Schritt 2)

Bemerkung:
- Schritt 1)&2) meistens sehr einfach
- Sehr häufig ist k=2, dann ist es günstig wenn die Teilprobleme die Größe n/2 haben (=Teile in 2 Hälften)
- Schritt 3) (Misch-Schritt) leistet die meiste Arbeit

Laufzeit von Divide and Conquer Algorithmen
Kann man durch eine rekursive Gleichung beschrieben werden. Sei T(n) die Laufzeit für Problem der Größe n

$$T(n)=\sum_{i=1}^{k}T(n)-T_{Teile}(n)+T_{Mischen}(n)$$

wobei: n_i Größe von Teilproblemen i ($1\leq i\leq k$)

Beispiel: Sortierung durch Mischen (Merge Sort)
Problem: (Vorgriff auf Kapitel III)
Feld: A[1..n] von beliebigen Zahlen
Aufgabe: sortiere dieses Feld Aufsteigend d.h. permutiere die Elemente so, dass dann $A[i]\leq A[i+1]$ für i=1..n-1
Beobachtung: 2 bereits sortierte Felder $A[1..{}^{n}/_{2}]$ $B[{}^{n}/_{2}]$ können effizient zu einem sortierten Feld C[1..n] zusammengemischt werden.

A:

1	7	12	17	25	30

i lesen

B:

3	7	10	17	20	24

j lesen

Min(A[i], B[j])

schreiben

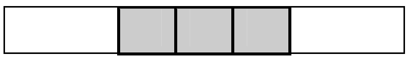
C:

1	3	7	7	10	12	17	17	20	24	25	30
i++	j++	i++	j++	j++	i++	i++	j++	j++	j++	i++	i++

Idee rekursiver Algorithmus

MergeSort(l,r) sortiert das Teilfeld

A[l,r]



1 l r n

1. MergeSort(l,r)
2. IF($l \geq r$) return; // Feld der Größe 1 oder 0 (Verankerung)
3. $m \leftarrow \frac{l+r}{2}$; // Teile
4. MergeSort(l,m) // Beherrsche
5. MergeSort(m+1,r) // 2. rekursiver Aufruf
6. Merge(l,m,r); //Misch Schritt Sortierung siehe Beobachtung in Feld C und wieder in Feld zurückkopieren

//END

Merge → Laufzeit O(n) Linear
Laufzeit von MergeSort auf Feld A[1..n]
Aufruf MergeSort(1,n)

$T(n) \leq 2\,T(^n/_2)$ + c_1 + $c_2\,n$ für Konstante c_1 und c_2

Beherrsche — Teile O(1) — Merge O(n) für N>1

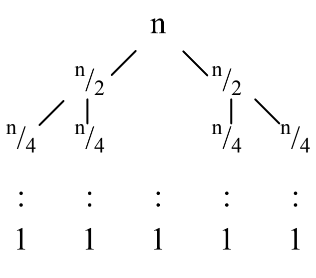
$T(n) \leq 2\,T(^n/_2) + c \cdot n$ [$T(n) = c_0$ für n=1]

Für eine Konstante c (abhängig c_1 und c_2 und n>1)

$T(n) \leq 2\,T(^n/_2) + c \cdot n = T(n) \leq 2\,T(^n/_2) + O(n)$

Induktiv Rekursionsbaum:

Höhe =$\log_2(n)$



Vermutung: Gesamtaufwand: $O(n \cdot \log(n))$

A.D Übung

MergeSort - Sortieralgorithmus

Laufzeitabschätzung für das Sortieren n Schlüssel)

Rekursionsgleichung: $T(n)=2\cdot T(^n/_2) + c_1\cdot n$, c_1 Konstante $c_1>1$

Frage: Wie löst man diese rekursive Gleichung?

1. Substitutionsmethode

   Idee: Rate die Lösung und verifiziere sie

   Bsp.:

$$Z(n) = \begin{cases} \Theta(1), n = 1 \\ 2 \cdot T(\frac{n}{2}) + c_1 \cdot n, n > 1 \end{cases}$$

Rate: $T(n)=O(n\cdot\log(n))$

Überprüfe Lösung durch Induktion

Zu zeigen: $T(n)\leq c\ n\cdot\log(n)$ für $c>0$

I.A Schranke hält $^n/_2$, d.h. $T(^n/_2)\leq c\cdot{}^n/_2\cdot\log(^n/_2)$

Einsetze in die Rekursionsgleichung:

$T(n) = 2\cdot(c\cdot{}^n/_2\cdot\log(^n/_2)+c_1\cdot n$

$\leq c\cdot n\cdot\log(^n/_2)+c_1\cdot n$

$= c\cdot n\cdot\log(n)-c\cdot n\cdot\log(2)+c_1\cdot n$

für $c > 1$ und $c \geq c_1$ erhalten wir $T(n) \leq c\cdot n\cdot\log(n)$

2. Integrationsmethode:

   Idee: Integriere die rek. Gleichung bis zur Verankerung

   Bsp.:

$$Z(n) = \begin{cases} \Theta(1), n = 1 \\ 3 \cdot T(\frac{n}{4}) + n, n > 1 \end{cases}$$

$$T(n) = 3 \cdot T(\frac{n}{4}) + nn, n > 1$$

$$= n + 3 \cdot \left(\frac{n}{4} + 3 \cdot T\left(\frac{\frac{n}{4}}{4} \right) \right)$$

Integriere wie folgt: $= n + 3^1 \cdot \frac{n}{4} + 3^2 \cdot T(\frac{n}{4})$

$$\leq n + 3^1 \cdot \frac{n}{4} + 3^2 \cdot T(\frac{n}{4^2})$$

$$\leq n + 3^1 \cdot \frac{n}{4} + 3^2 \cdot \frac{n}{4^2} + 3^3 \cdot \frac{n}{4^3} + \ldots + 3 \cdot \log_4(n) \cdot \Theta(1)$$

$$\leq n \cdot \sum_{i=0}^{\infty} (\frac{3}{4})^i + n \cdot \log_4(3)$$

a) $n \cdot \sum_{i=0}^{\infty} (\frac{3}{4})^i = n \cdot \left(\frac{1}{1 - \frac{3}{4}} \right)$

b) $3 \cdot \log_4(n) = (4 \cdot \log_4(3)) \cdot \log_4(n)$

$= (4 \cdot \log_4(n)) \cdot \log_4 3 = n \cdot \log_4(3) \leq n$

aus (a)+(b) erhalten wir T(n)≤5·n=O(n)

$$\sum_{i=0}^{\infty} x^k = \frac{1}{1-x}, |x|<1$$
geometrische Reihe

Wichtige Summenformeln:

Arithmetische Reihe: $\sum_{k=1}^{n} k = \frac{1}{2}\cdot(n+1) = \Theta(n^2)$

Geometrische Reihe:
$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}, x \neq 1$$
$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}, |x|<1$$

Harmonische Reihe: $H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + O(1)$

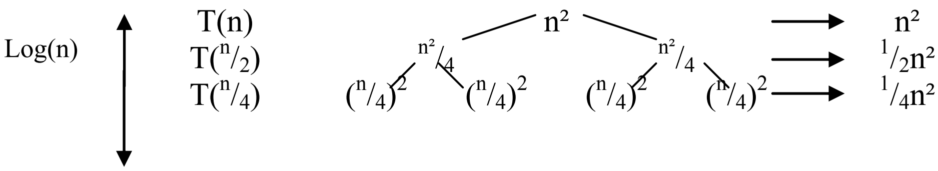
Integrierende Reihe: $\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}, |x|<1$

$$\left(\sum_{k=0}^{\infty} x^k\right)' = \left(\frac{1}{1-x}\right)'$$

Folgt aus geometrische Reihe:
$$\sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$$
$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$
$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$

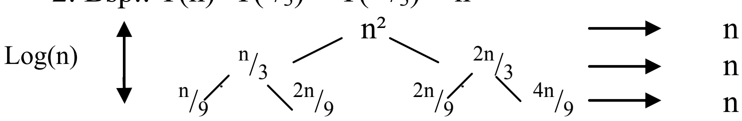
Rekursionsbaum als Hilfsmittel bei der Integrationsmethode

1. Bsp.: $T(n) = 2 \cdot T(^n/_2) + n^2$



$$\Theta\left(\sum_{T=0}^{\log(n)} \frac{1}{2^T} n^2\right) = \Theta(n^2)$$

2. Bsp.: $T(n)=T(^n/_3) + T(^{2n}/_3) + n$



$$\sum_{i=0}^{\log(n)} n = \Theta(n \cdot \log(n))$$

2. Einfache Datenstrukturen

2.1. Felder

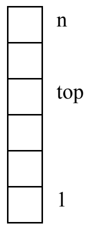
(Array) A[1..n]

2.2. Keller (Stack)

a) Beschränkter Größe (maximale Größe n)

Feld A[1..n]

Index; Top $\in$ {0..n}



Nur Zugriff auf das oberste Top-Element Operationen auf dem Stack S

(Elemente des Stacks sind alle vom gleichen Typ)

S.clear():	Top $\leftarrow$ 0;
S.empty()	return Top=0;
S.push(x)	A[++Top] $\leftarrow$ x;
	(Problem: Overflow; top>n)
S.top()	return A[Top];
	(Problem: Top=0)
S.pop()	return A[Top--];
	(Problem: leerer Stack)

b) unbeschränkter Größe

d.h. beliebige Größe

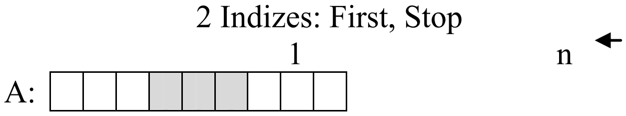
Stacks nennt man auch LIFO-Queue (Last in First-Out)

2.3. Schlangen (Queue)

c) Beschränkte (maximal Größe n)

Feld A[1..n+1] (Größe n+1)

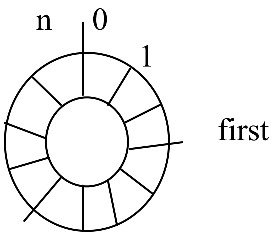
2 Indizes: First, Stop



First	Stop
(links)	(rechts)
Vorne	Hinten
Vorne	Anfügen/
Wegnehmen/	append
löschen pop	
Lesen top	

Zyklische Sicht eines Feldes A[0..n] (Größe n+1)

Schlange Q:



stop

leer: stop=first
(zur Unterscheidung von Komplett gefüllter Schlange verwenden wir Feld der Größe n-1)
Operationen:

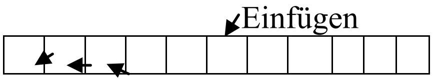
Q.clear()	stop←0; first←0;
Q.empty()	return stop=first;
Q.append(x)	A[stop]←x;
	Stop←(stop+1) mod (n+1)
Q.top()	return A[first] ;
Q.pop()	y←A[first] ;
	First←(first+1) mod (n+1)
	Return y ;

Probleme: Overflow (append) ; unterflow (top)
Um die Exzeption abzufangen verwendet man einen Zähler der Elemente (→count)
Overflow: if (count>n)
Unterflow if (count<n)
Alle Operationen brauchen O(1) Laufzeit
Speicherplatz O(n)
Schlangen heißen auch FIFO
(First-In-First-Out)
Erweiterung (Variante)
Man kann auf beiden Seiten (vorn + hinten)
Elemente hinzufügen und entfernen. Die entsprechende Datenstruktur
DEQueue
DoubleEndeQueue

d) Unbeschränkte d.h. beliebige Größe

2.4. Listen

Folge von Elementen, die man dynamisch verwalten kann
Einfügen an beliebiger Position $\Big\}$ O(1)
Löschen an beliebiger Position
Problem bei Feldern

A: 

Kostet O(n)

Dynamische Datenstruktur
Strukturen (Klassen, Pointer, Referenzen)
Bsp. Class Klasse XYZ

```
{
  Int x; int y; int z;          //Dateneinträge
}                               //Verbund
```

Wert (Objekt)

a

		x	y	z			

Zugriff auf Daten-Einträge (Member)

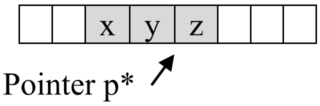
Syntax x,y,z ∈ a   statisch Laufzeit-Stack des Programms

a.x←1;
a.y←2;
a.z←3;

Dynamische Objekte & Pointer (Referenz)

Auf Laufzeit-Heap

new xyz;
xyz*p new xyz;



Pointer p*

Zugriff *p→a ;	Abkürzung
(*p).x←1 ;	p→x←1 ;
(*p).y←2;	p→y←2;
(*p).z←3;	p→z←3;

Freigabe von dynamisch angebenen Objekt
Xyz * p
Delete p;
Danach ist jeder Zugriff auf p illegal!

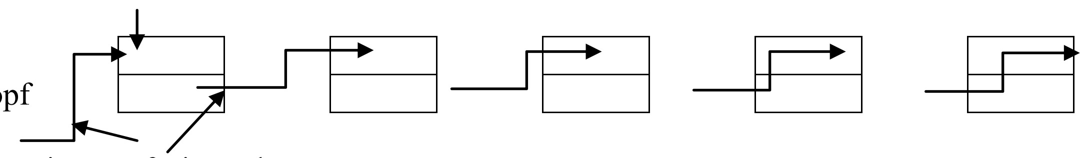
Null-Pointer:
Xyz *p;   p←Null;
Symbol :   ⊥

a) Einfach Verkette Listen

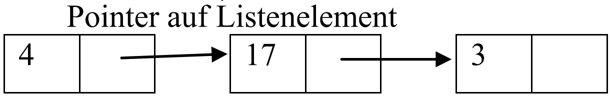
Listen Elemente

Daten

Kopf



Pointer auf Listenelement




```
Class : SListElem() //Simply indirect List
{
    Int Inhalt ;
    sListElem* next ;
}
```

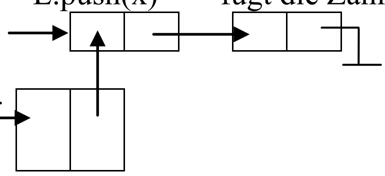
Eigentliche Liste wird realisiert durch Pointer auf 1. Element
sListElem* Kopf←NULL; //Leere Liste

Operationen auf Liste L;

- Initialisierung Kopf←Null;
- L.push(x)   fügt die Zahl x als neues erstes Element in die Liste ein

Kopf

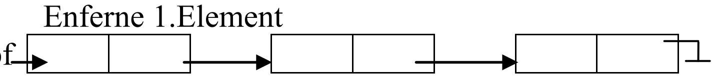
Neuer Kopf



p→Inhalt←x;
p→next←Kopf;
Kopf←p;

L.Top()   return Kopf→Inhalt;
L.empty()   return Kopf=NULL;
L.pop()
Enferne 1.Element

Kopf

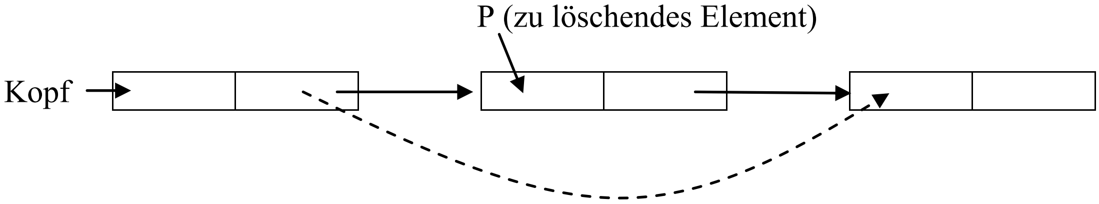


P←Kopf; (Kopf≠Null)
Kopf←p→next;
Delete p; //Speicherfreigabe
Interation: forall x in L do … od;
P←Kopf
While p≠Null
Do
x←p→Inhalt
$\boxed{}$ Rumpf
p←p→next
od

Variante: Zusätzlicher Zeiger auf letztes Element
Zusätzliche Operationen L.append(x)
Allgemein: Falls p Pointer auf beliebiges Listen Element ist dann kann man unmittelbar nach p ein neues Element (effizient) einfügen bzw. löschen

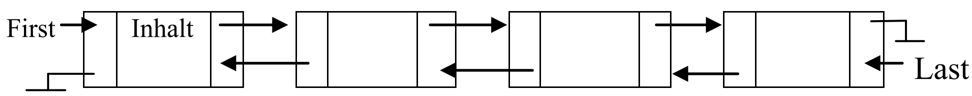
Laufzeit: Alle Operationen in O(1)
Platzbedarf O(n) wenn n Länge der Liste

Einfach verkettete Listen unterstützen das Entfernen eines geben Elements nicht effizient



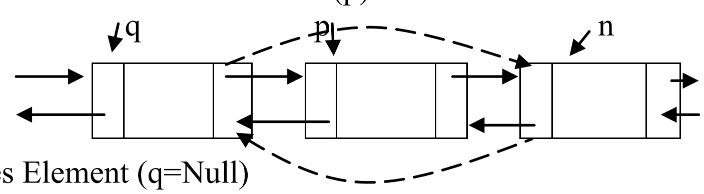
Wir benötigen einen zweiten Pointer auf den Vorgänger (Predesseser)

b) Doppelt verkette Listen (Double Linker Lists)



class dListElem
{
Int inhalt;
dListElem* next;
dListElem* prect;
}

Operationen:
Alle von einfach verketten Listen + L.remove(p)



Sonderfälle: Erstes Element (q=Null)
Letztes Element (n=Null)

⇨ Evtl. Neue Werte für First und Last
L.insertAfter(p,x) auch mit sList

L.insertbefor(p,x) nur mit dList

Auch Interation in umgekehrter Reihenfolge möglich
Alle Operationen Zeit: O(1)
Speicherplatz O(n)

3. Sortieren

3.1.1. Allgemeine Sortierverfahren

Beruhen auf vergleichen
Problem gegeben Menge S aus einem Wertebereich U (auch Universum genannt), auf dem eine lineare Ordnung „≤“ definiert ist Beispiel: U=*Z* (int), „≤“: „normale ≤ auf Zahlen
U=∑*, ≤ textographische Ordnung
Aufgabe Bringe die Elemente von S in eine aufsteigende Reihenfolge bzgl. ihrer linearen Ordnung ≤
Konkreter: S ist als Feld A[1..n] (n Elemente)
Aufgabe: Ordne das Feld so um, dass A[i]≤A[i+1] für 1≤i≤n-1
Variante: die meistens auch in der Praxis vorkommt:
S ist Menge von Objekten (Datensätze) wird jedem x ∈ S ein Schlüssel Key(x) ∈ U zugeordnet (z.B. Matrikelnummer, Name, Alter…)
Sortiere: S bezgl. Dieses Schlüssels aufsteigend.
Ab jetzt U=*Z* (int) mit „≤“
Bemerkung: Test auf (Un)Gleichheit ist immer diffiniert if (x=y) if (x≠y) ➔ Test auf <, >, ≥

3.1.2. Allgemeine Sortieralgorithmen

Sortieren durch Maximumsauswahl
Intuitives Verfahren:
Eingabe Feld A[1..n] von ganzen Zahlen
Idee: 1) Suche das max. Element →A[j]
2) vertausche A[j] mit A[n]
3) wiederhole 1) und 2) mit A[1..n-1] bis Feld sortiert ist
Algorithmus (Pseudo-Code)

1. r←n;
2. *while* r>1
3. *do* //Maximum in A[1..n]
4. j←1;
5. *for* i=2 *to* r *do*
6. *if* (A[i]>A[j]) *dann* j←i ; *fi*;
7. *od* ;
8. A[j]←→A[r] //swap
9. r-- ;
10. *od* ;

Laufzeit (Sortierung durch Maximums auswähl)
Innere Schleife : Lineare Suche nach Max A[1..r]
Kostet O(r) Zeit
Wird ausgeführt für r=n,n-1n-2,…,2

Gesamtlaufzeit: $O\left(\sum_{i=1}^{n} i\right)$ arithmetische Reihe

$^{1}/_{2}$ n (n-1) $= {}^{1}/_{2}(n^2+n)$

$= O(n^2)$

⇨ Verdoppelung der Eingabe n bewirkt Vervierfachung der Laufzeit

Problem: Die Resultate der Vergleiche (maximal O(r)), die der Algorithmus ausführt um das #Maximum zu finden, werden bei der nächsten Maximum Suche nicht wieder verwendet (d.h. gehen verloren)

Frage: kann man Maximum schneller finden?

Antwort: Ja! Mit einer Datenstruktur, die auf einen gewissen Art die Resultate der Vergleiche speichert

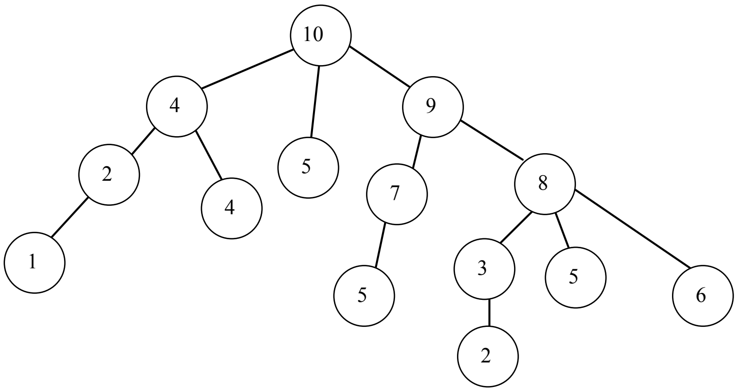
➔ Heapsort

Datenstruktur: Heap (Haufen)

2 Arten (Maximum Heap, Minimum Heap

Def: Ein Heap ist ein Baum dessen Knoten mit Zahlen (Schlüssel beschriftet sind, so dass gilt. Für jeden Knoten v (≠Wurzel) ist Zahl(v) ≤ Zahl(Vater(v))

Bsp:

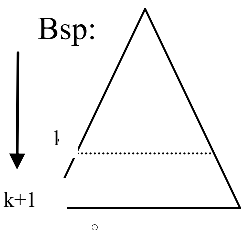


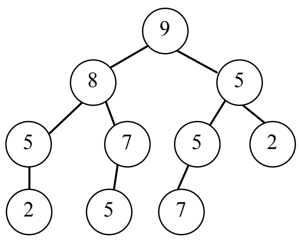
Folgerung: Wurzel des Heap enthält das Max

Definition ausgeglichene Heaps

a) Binär: alle Knoten haben 2 oder 0 Kinder für evtl. einen Knoten mit einem Knoten
b) ausgeglichen:
    1) Es gibt eine Tiefe k, so dass alle Blätter auf Tiefe k oder k-1 liegen
    2) Auf Tiefe k+1 stehen alle Blätter möglichst weit links

Bsp:





Realisierung eines Heaps als Feld A[1..n]

Idee: Speichere die Elemente (Zahlen) des Heaps Level für Level in Feld A

1	2	3	4	5	6	7	8	9	10
10	8	5	5	7	5	2	2	5	7
Wurzel									

1) Kinder eines Knoten i sind 2i (links), 2i+1 (rechts) Existiert nur falls < n
2) Knoten i ist ein Blatt, wenn 2i>n

3) Vater von i hat die Nummer $\lfloor i/2 \rfloor$ (für $i \neq 1$)

Definition Heap:

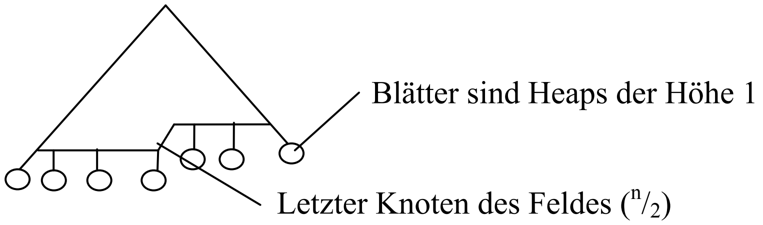
Ein Feld A[1..n] heißt Heap, wenn für alle i=2,3..,n gilt $A[\lfloor i/2 \rfloor] \leq A[i]$

Heap Sort besteht aus zwei Phasen

1. Aufbauphase: Verwandelt das Eingabe Feld A[1..n] in ein Heap
2. Selektionsphase: Verwendet Heap zum Sortiern

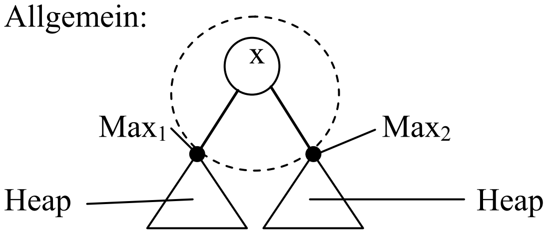
1) Aufbauphase:

Beobachtung: Baum mit nur einem Knoten (Blatt) ist immer ein Heap



Idee: Konstruiere aus 2 Heaps kleinerer Höhe einen Heap (größerer Höhe)

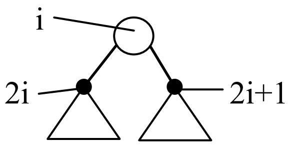
Allgemein:



Hier könnte die Heap-Eigenschaft verletzt sein d.h. $x<max_1$ oder $x<max_2$

Algorithmus: Sink(i,n) Heapify

Arbeitet auf Feld[1..n] mit Kinder i (2i, 2i+1) sind Wurzeln von Heaps d.h. Unterbäume von i sind Heaps



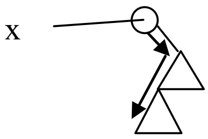
Aufruf Sink(i, n)
Verwandelt den Teilbaum mit Wurzel i in einen Heap

Arbeitsweise:

1. $A[i] \leftarrow \max(A[i], A[2i], A[2i+1])$
   Bestimme Max von A[2i] und A[2i+1]
   Vergleiche A[i] mit diesem Max A[j]
   Falls A[i]<A[j] vertausche A[i] mit A[j]
2. Ruf Sink(I,n) rekursiv auf
   Verankerung, wenn $A[i] \geq Max(A[2i],A[2i+1])$

Veranschaulichung:

Sink(i, n)
Lasse x im Heap herunter sinken



Aufbauphase:

Beim letzten Knoten der Kinder hat ➔ $i={}^n/_2$ ruf sink (i, n) auf

Und geht dann rückwärts bis die Wurzel

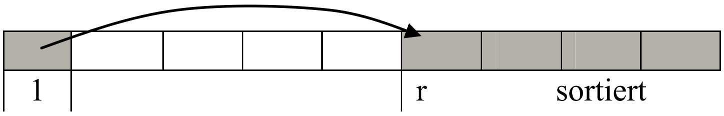
Ruft sink auf

For $i={}^n/_2$ downto 1 do

Sink(i, n)

od;

Selektionsphase: Sortierung durch Maximumsauswahl

A: 

Hier A[1..r] ist Heap => Max A[1]
Vertausche A[1] mit A[r]
Vermindere r um 1
Verwandle A[1..r] in einen heap durch sink

1. r←n
2. WHILE r>1
3. DO
4.   A[1]←→A[r];
5.   r←r-1;
6.   Sink(1, r)
7. OD;

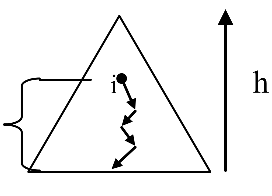
Sink(i,n)

1. x←A[i];
2. j←2i; // linke Seite
3. WHILE (j≥n) // i kein Blatt
4. DO
5.   IF (j<n) THEN
6.   IF (A[j]<A[j+1]) THEN j←j+1;
7.   FI;
8.   IF (x<A[j]) THEN
9.   A[i]← A[j]
10.   i←j;
11.   j←2i;
12.   ELSE break; //x richtige Position
13. DO;
14. A[i]←x;

Laufzeitanalyse:

1. Ein Aufruf Sink(i ,n) auf A[1..n]

Heap:
Höhe



Binär und ausgeglichen Höhe log n
Einfache Laufzeitschranke O(Höhe) =O(log n) (worst case)

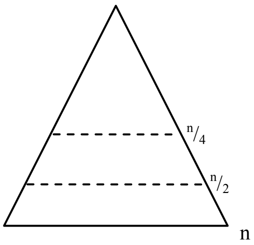
Genauer Betrachtung liefert Sinkt(i, n) hat Laufzeit O(Höhe(i))
Bemerkung: Wichtiges Komplexitätsmaß beim Sortieren ist die Zahl der Vergleiche
Allgemein: O(1) pro Knoten (Level, Höhe) ➔ max. 2·Höhe(i)

Aufbauphase: FOR $i={}^{n}/_{2}$ downto 1 DO
    Sink(i, n)
OD;

Laufzeit:

$$O\left(\sum_{i=1}^{n/2} Höhe(i)\right)$$

$$= O\left(\sum_{n=0}^{\log n} h \cdot \underbrace{(\#Knoten\ \ auf\ \ Höhe(h)}_{\leq \frac{n}{2^h}}\right)$$



$$= O\left(\sum_{n=0}^{\log n} h \cdot \frac{n}{2^h}\right) \leq O\left(h \cdot \sum_{n=0}^{\infty} \frac{h}{2^h}\right)$$

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2} \qquad , x < 2$$

Integrierende Reihe

Sink $x = \frac{1}{2}$

$$O\left(n \sum_{i=0}^{\infty} \frac{h}{2^h}\right) = O\left(n \frac{1/2}{(1 - 1/2)^2}\right)$$

$$= O\left(n \cdot \frac{1/2}{n/4}\right) = O(2n) = O(n)$$

⇨ Aufbauphase hat eine Laufzeit: O(n)

2) Selektionsphase

```
FOR r=n downto 2 DO
    A[1] ←→ A[r]        //O(1) → O(n)
    Sink(1, r-1)        //O(Höhe von 1 bis r-1) =O(log r)
OD
```

Sink(1, r-1)-Aufrufe:

$$O\left(\sum_{r=2}^{n} \log r\right) \leq O\left(\sum_{r=1}^{n} \log n\right) = O(n \cdot \log(n))$$

Sektionsphase O(n+n log(n))=O(n log(n))

Satz: Heapsort auf einem Feld der Länge n hat Laufzeit O(n log n)

(gilt auch für die Anzahl der Vergleiche)

Beweis; Aufbauphase: O(n)

Selektionsphase O(n log(n))

O(n log(n))

Bemerkungen:

- Heapsort ist ein In-Place-Sortierverfahren d.h. arbeitet nur auf dem Eingabefeld A[1..n] ohne zusätzlichen Speicher (außer O(1) für Variable
  Im Gegensatz zu MergeSort (braucht Temp-Feld)
- Heapsort nutzt den Cache (internen kleinen schnellen Speicher) besser wie andere Sortierverfahren (Mergesort, Quicksort (viele große Sprünge im Feld)

3.1.3. Allgemeine effiziente Sortieralgorithmen

Quicksort: Sortien durch teilen (Divide & Conquer)
Idee: A[1..n] Eingabe

1) Wähle beliebiges Element $x \in A$ (z.B $x \leftarrow A[1]$) = das Pivot Element

2) Ordne das Element von A so um, dass zuerst alle Elemente <x im Feld stehen

dann x und dann alle Elemente >x folgen (durch vertauschen auf A)

<x	x	>x

A1 A2

X bereits korrekte Position im Sortierten Feld

Diesen Schritt 2 nennt man Partitionieren

3) Wende den Algorithmus rekursiv auf die Teilfelder A1 und A2 an

Verankerung für $n \leq 1$ ist nichts zu tun

Divide & Conquer
Die Arbeit wird in Teil-Schritte (Partitionieren)
Beherrsche (Misch)-Schritt ist trivial
(umgekehrt wie bei Mergesort)
Implementierung der Partitionieren auf Feld A[1..n]
(allgemein Teilfeld A[l..r])

x	<	≥		<	≥
	↑i	↑i		↑j	j↑

Partitionieren:

```
i←2;
j←n;
x←A[1];
REPEAT
        WHILE (i<n && A[i]<x) i++
        WHILE (j<n && A[j]<x) j++
        IF (i<j) THEN
                A[i]←→A[j];
                i++;
                j--;
        FI;
UNTIL; i>j;
```

x	<x		>x
		j i	

A[1]←→A[j]

Beispiel:

42	55	12	44	94	6	18	67
	i						j
42	18	12	44	94	6	55	67
		i					j
42	18	12	6	94	44	55	67
6	18	12	**42**	94	44	55	67

Rekursive Quicksort – Funktion
Eingabe A[1..n] von Zahlen

```
Quicksort(l, r)                // sortiert A[l..r]
IF (l ≥r-1)  return;           // Verankerung weniger als 2 Elemenente
x←A[l];
i←l+1;
j←r;
REPEAT
   WHILE (i < r && A[i] < x) i++;
   WHILE (j > l && A[j] ≥ x) j--;
    IF (i < j) THEN
       A[i]←→A[j]
       i++; j--;
    FI;
UNTIL i > j;
 A[l]←→A[j]
Quicksort(l,j-1);              //rekursive Aufrufe
Quicksort(j+1,r);              // für A1 und A2
```

Analyse der Laufzeit:
Partinierung auf einem Feld der Länge n hat Laufzeit
O(n) (Teile Schritt)
Beherrsche Schritt trivial Laufzeit O(1)
Allgemein Divide & Conquer
$T(n) = T(|A1|) + T(|A2|) + T_{Teile}(n) + T_{Beherrsche}(n)$
⇨ $T(n) = T(n_1) + T(n_2) + O(n)$
$(n_1 = |A1| \;\; n_2 = |A2|$
Im Algorithmus für l = 1 r=n
$n_1 = j – 1$

$\overbrace{\quad n_1 \quad}^{j-1}$	x	$\overbrace{\quad n_2 \quad}^{n-j}$

$T(n) = T(j – 1) + T(n-1) + O(n)$
j hängt von der Eingabe ab
Wir unterscheiden folgende Fälle
1. Laufzeit im besten Fall
Partitionierung setzt Pivot-Element x immer in die Mitte also $j={}^n/_2$ für die Eingabe
⇨ $T(n) \le 2\,T({}^n/_2) + O(n) = O(n \log(n))$ siehe Mergesort
Rekursion ist dann perfekt balanciert

2. Laufzeit in schlechtesten Fall

Total unbalancierter (djungierte) Rekursion

$n_1=0$ $n_2=n-1$

Mögliche Eingabe: aufsteigend sortiertes Feld
Absteigend sortiertes Feld

T(n) = T(n-1) + O(n)

=T(n) + T(n-1) + T(n-2) + … + T(1)

=O(n²)

Vermeidung des schlechtesten Fall (Laufzeit n²)

Zufällige Wahl des Pivot Element

⇨ Randomisierter Algorithmus

Z←random(l, r); // Zufallszahlgenerator

A[l]←→A[z]; // liefert $i \in$ [l..r] mit prob= $^1/_n$ (wenn n = r-l+1

Mit großer Wahrscheinlichkeit wird der schlechste Fall vermieden (unabhängig von der Eingabe)

Zufällige Permutation der Eingabe

⇨ Jede Permutation aus n! möglichst gleich wahrscheinlich

3. Laufzeit im mittleren Fall (erwartete Laufzeit)

Annahme: 1. Alle Zahlen in A[1..n] sind paarweise verschieden
2. Jede der n! Permutationen der Eingabe ist gleich wahrscheinlich

Ohne Beschränkung der Allg.: Zahlen = {1,…n}

Folgerung: Für $1 \le k \le n$ gilt prob(A[1]=k)=$^1/_n$

Sei $\overline{T}(n)$ die erwartete Laufzeit von n unter der Annahme genauer die erwartete Zahl der Vergleiche

$\overline{T}(0)=\overline{T}(1)=0$ // Zahl der Vergleiche

$\overline{T}(n)=n+$ „Erwartungswert von T(A1) + T(A2)"

$$=n+\frac{1}{n}\sum_{j=1}^{n}\overline{T}(j-1)+\overline{T}(n-j)$$

Abschätzung von $\overline{T}(n)$

$$\overline{T}(n)=n+\frac{2}{n}\sum_{j=0}^{n-1}\left(\overline{T}(j)\right) \qquad |\cdot n$$

$$n\cdot\overline{T}(n)=n+2\sum_{j=0}^{n-1}\overline{T}(j) \qquad |n\to n+1 \qquad (*)$$

$$(n+1)\cdot\overline{T}(n+1)=n+1+2\sum_{j=0}^{n}\overline{T}(j) \qquad (**)$$

$$(n+1)\cdot\overline{T}(n+1)-n\cdot\overline{T}(n)=(n+1)^2-n^2+2\overline{T}(n) \quad (**-*)$$

$$(n+1)\cdot\overline{T}(n+1)=2\cdot n+1+(n+2)\cdot\overline{T}(n)$$

$$\overline{T}(n+1)\le 2+\frac{n+2}{n+1}\overline{T}(n)$$

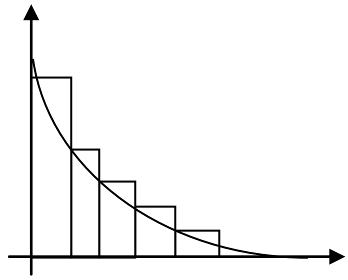
Skizziere: $=2+\frac{n+2}{n+1}\left(2+\frac{n+1}{n}\left(2+\frac{n}{n-1}(...)\right)\right)$

$$=2+(n+2)\left(\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+...+1\right)$$

$$= 2(1 + (n+2)(\frac{1}{n+1} + \frac{1}{n} + ... + \frac{1}{2})$$

$$\Rightarrow \overline{T}(n) \le 2(1 + (n+1)\sum_{j=2}^{n} \frac{1}{j} - 1$$

$$= 2(1 + (n+1)\sum_{j=1}^{n} \frac{1}{j} - 1$$



Es gilt:

$$\sum_{j=1}^{n} \frac{1}{j} \le 1 + \ln n$$

$$\le 1\int_{1}^{n} \frac{1}{x} dx$$

$$= 1 \ln n$$

$$\overline{T}(n) = O(n \ln(n))$$

$$= O(n \log(n)) \qquad \text{da} \log n = c \cdot \ln n$$

Satz: Quicksort auf Feld der Länge n hat Laufzeit
a) $O(n^2)$ im schlechtesten Fall
b) O(n log n) im mittleren Fall
Bemerkung: Wenn die Eingabe gutartig (d.h. zufällig) dann ist Quicksort der schnellste Sortieralgorithmus
Gründe: in-Place (wie Heapsort)
Gute Lokalität (wie Mergesort)

Definition: Stabilität
Ein Sortieralgorihmus heißt stabil, wenn Objekte mit gleichem Schlüssel in der gleichen Reihenfolge ausgegeben werden, wie in der sie Eingabe erscheint
Wichtig für das Sortieren nach mehren Kriterien (Schlüsseln)
Bsp.: Studierende der Uni, sollen sortiert werden nach Alter und Matrikel-Nr.
Alter: Primäre (wichtigster) Schlüssel
Matrikel-Nr.: sekundäre Schlüssel
d.h. bei gleichem Alter nach Matrikel-Nr. sortieren
Beobachtung:
1. Reihenfolge: Sortieren nach Matrikel-Nr. dann nach Alter d.h. das wichtigste Kriterium am Ende
2. Stabilität für 2. Sortierlauf (nach Alter) notwendig, denn bei gleichem Alter soll Sortierung des 1. Laufs erhalten bleiben

3.2. Spezielle Sortierverfahren

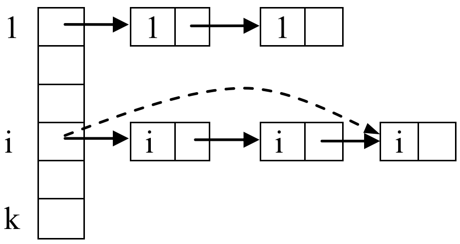
Neben spezielle Eigenschaften der Schlüssel
Hier in der Vorlesung: Schlüssel sind ganzen Zahlen aus einem steigenden Intervall (z.B. Alter, Matrikel-Nr., Buchstaben)
Genauer Schlüssel [{1,2,…,k} für ein festes k.
Vereinfachtes Problem (wir betrachten nur die Schlüssel) Sortier ein Feld A[1..n] von Zahlen {1..k} aufsteigend

3.2.1. Bucketsort

Sortierung durch Fachverteilung
Idee:

- Stelle k leere Fächer bereit
- Durchlaufe das Feld A und füge das jeweilige Element x (Schlüssel) $\in \{1..k\}$ in das jeweilige Fach mit Nummer x
- Gehe Fächer 1 bis k durch und gib die Inhalte aus

Realisierung (Datenstruktur):
Fächer: Feld B[1..k] von einfach verketteten Listen



Bucket Sort

```
1.  FOR i=1 to k DO B[k].clear OD;
2.  FOR j=1 to n DO
3.    x←A[j];              // x←key(A[i])
4.    B[x].append x        //B[x(Schlüssel)].append A[i](Objekt)
5.  OD;
6.  L←0;
7.  FOR i=1  to k DO          }
8.    FORALL x in B[i] DO     }   Aufsammeln
9.      A[++i]←x OD;          }
10. OD;                       }
```

Laufzeit:

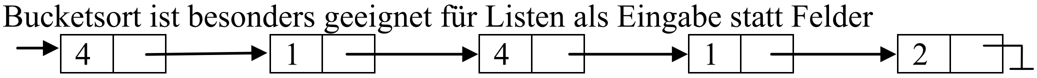
1. Initialisierung (Zeile 1): O(k)
2. Verteilen (Zeile 2-5): O(n)
3. Aufsammeln (Zeile 6-10): O(n + k)

Insgesamt: O(k + n)

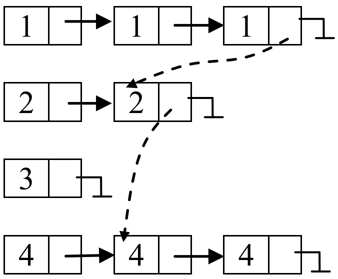
Für k=O(n) kann man in hier in Linarzeit O(n) sortieren.
Bemerkung
1. Algorithmus ist stabil (wegen „append“)
2. Stabilität ist auch möglich ohne Pointer auf letztes Listenelement (→append)

```
FOR i=n downto 1 DO
    X←A[i]
    B[x].push(x)
OD
```

3. Bucketsort ist besonders geeignet für Listen als Eingabe statt Felder

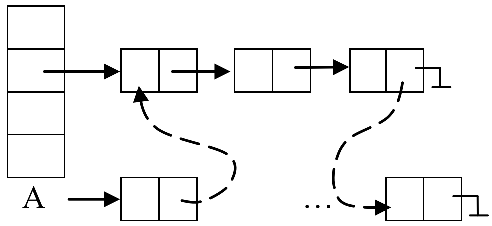
A: 

➔ k=4
Dann kann man die Listen Elemente der Eingabe direkt in die Buckets anhängen.



Auf Listen
Aufsammeln konkateniere alle Listen B[1]..B[k]
A.Clear() ist schon leer nach verteilen

```
FOR i=1 to k DO
  A.conc(B[i]);
OD
```



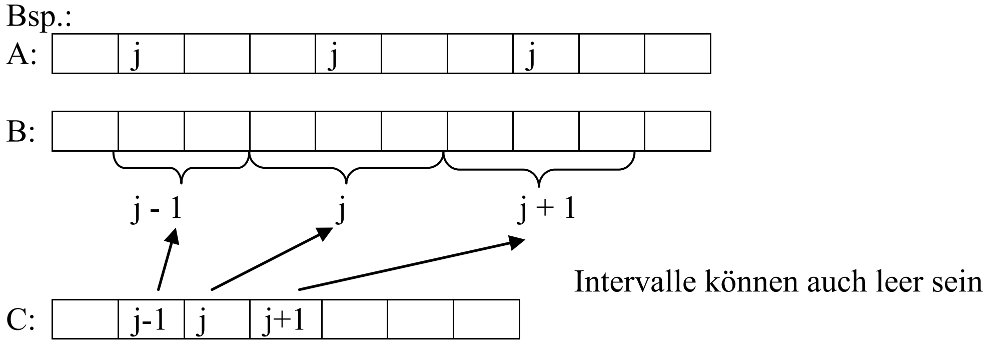
Für Listen benötigt Bucketsort nur das Feld B[1..n] von Listen als zusätzlichen Speicher d.h O(k) zusätzlichen Platz
Für Felder zusätzlich Platz O(n+k)
Bemerkung: Listenverwendung ist auch besonders gut geeignet für mehrfach Verwendung von Bucketsort mit verschiedenen Schlüsseln

3.2.2. Counting Sort (Sortierung durch Zählen)

- Löst gleiches Problem wie Bucketsort
- Ist gut geeignet für Felder
- braucht immer O(n + k) zusätzlichen Platz
- Stabil

Eingabe A[1..n] von Zahlen aus {1,..,k}
Ausgabe B[1..n] aufsteigend sortiert
Hilfsfeld C[1..n] (Count-Feld)

Idee: Berechne für jeden möglichen Schlüssel j ∈ {1..k} das Intervall im Ausgabe Feld B in das die entsprechenden Eingabe Werte geschrieben werden müssen.
Bsp.:



Berechne für j = 1..k C[j]=Position des rechten Schlüssel j in sortierten Feld
Berechne das C[j]
Zähle wie oft j in A vorkommt
Durchlaufe C und addiere die Ergebnisse aus Schritt 1 auf

Counter-Sort

```
1. FOR j=1 to k DO C[j]←0; OD
2. FOR i=1 to n DO C[A[i]]++ OD
3. FOR j=2 to k DO C[j]=C[j]+C[j-1] OD    //C[i]=Anzahl aller Eingabe Werte
5. FOR i=n downto 1 DO                     //rückwärts
6.       x←A[i];
7.       B[C[i]]←x;
8.       C[x]--;
9. OD
```

Bsp.: k=5 n=12

A:	3	5	4	3	5	2	3	1	2	1	2	5

C:	2	3	3	1	3	zählen
C:	2	3	3	1	3	aufaddieren

	1er		2er			3er			4er	5er		
B:	1	1	2	2	2	3	3	3	4	5	5	5

Ausgabe

```
FOR i=n to 1 DO
      x←A[x]
      B[C[x]]←x
      C[x]--
OD
```

Laufzeit von Countingsort

Initialisierung von C:	O(k)
Zählen:	O(n)
Aufaddieren:	O(k)
Ausgabe:	O(n)
Insgesamt:	O(n+k)

4. Datenstrukturen für Menge und Wörterbücher

Definition: Wörterbuchproblem:

Sei k eine Menge von Schlüsseln (z.B. K=*Z*) und I eine Menge von Informationen (z.B. alle Strings)

1. Ein Wörterbuch D mit Schlüsselmenge K und Informationsmenge I ist eine partielle Abbildung $d:K\rightarrow I$

Dom(d), der Definitionsbereich von D sind alle Schlüssel aus K denen Informationen zugeordnet werden.

Beispiel:

D: K=*Z* Matrikelnummer
I= Namen (aller Studierenden)

Telefonbuch:
K=Namen
I=Telefonnummern

Operationen auf einem Wörterbuch D

D.Lookup(k) für $k \in K$ {d(k), falls $k \in dom(d)$ sonst undef

D.Member(k) liefert { true $k \in dom(d)$ sonst false

D.Inset(k, i) modifiziert die Abbildung für $k \in K$ und $i \in I$

$d \rightarrow d'$

1) $dom(d') = dom(d) \cup \{d\}$

2) $d'(x) =$ { $d(x)$ falls $x \neq k$ i falls $x=k$

Bemerkung falls $K \in dom(d)$ wird die entsprechende Information überschrieben

D.Delete(k) modifiziert die Abbildung
$d \rightarrow d'$, so dass $dom(d') = dom(d) \setminus \{k\}$

D.Clear() $dom(d) \leftarrow 0$

Menge: Speichert Teilmenge von K (dom(d)) hat Operationen, member(k), insert(k), delete(d), clear()

Beobachtung: Datenstrukturen für Menge kann man leicht zu einer Datenstruktur für Wörterbuch erweitern.

Speichern in der Menge Objekt (Paare) aus $I^x K$ (analog zum sortieren)

4.1. Allgemeine Datenstrukturen für Mengen & Wörterbücher

Schlüsselmenge K ist linear geordnet, d.h. $\leq$ Relation ist definiert ($\rightarrow <, =, >, \neq, \geq$)

Die Datenstruktur verwenden ($\leq$)-Vergleiche

4.1.1. Binäre Suchbäume

1) Knotenorientierte binäre Suchbäume

Ab jetz zur Vereinfachung $K=Z$

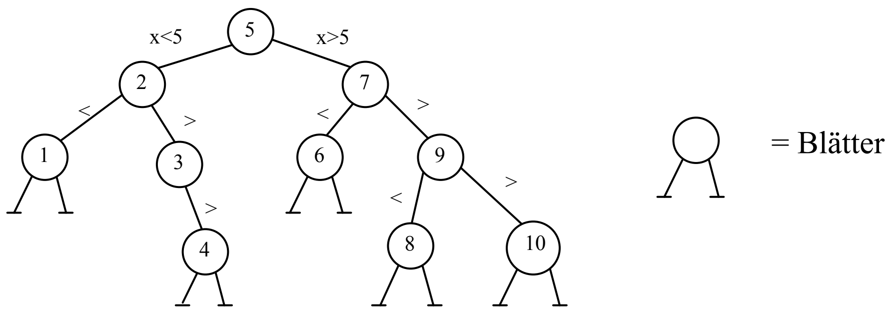
Problem: Verwalte eine Menge $S \leq Z$ unter den Operationen (member, insert, delete)

S wird in den Knoten eines Binären Baumes abgespeichert

Die Binäre Suche auf einem Feld, das die Elemente von aufsteigend sortiert sind

S=											
	1	2	3	4	5	6	7	8	9	10	Binär Such nach x
	l				m					r	

Binärer Baum



Rekursive Definition

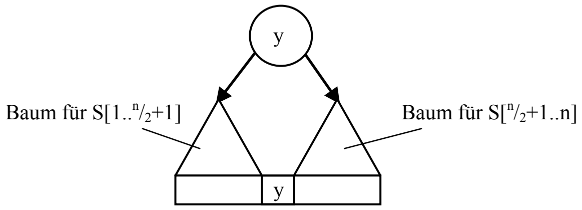
Ein Knoten-orientierter binärer Suchbaum für eine sortierte Folge S kann wie folgt definiert werden:

1) Falls S=0, dann der leere Baum (Keine Knoten)

2) Sonst a) Wurzel speichert das mittlere Element $y = A[m]$

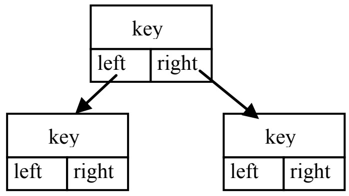
b) das linke Kind von r ist Wurzel eines Baumes für alle $x \in S$ mit $x<y$

c) das rechte Kind von r ist Wurzel eines Baumes für alle $x \in S$ mit $x>y$



Realisierung von Binären Baumen

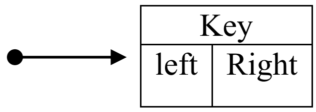
class Bin_tree_node
{
    int key;
    bin_tree_node* left;
    bin_tree_node right;
}

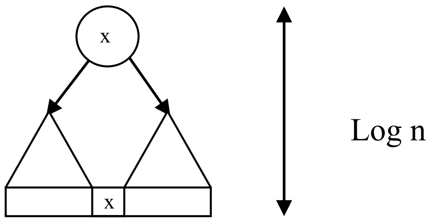


Übung Binär Baum(l, r)

Baut Knoten orientierten binär Baum für Teilfeld S[l, r] und liefert Pointer auf Werte

NULL l > r



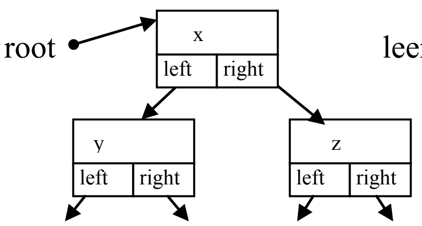


⇨ Ausgeglichen/ balanciert ➔ höhe ist log n

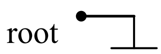
Suche in einem Knoten-orientieren-binären Baum

Search(x)   liefert Pointer auf Knoten v mit v.key=x
             (NULL-Pointer, falls x nicht im Baum)

Datenstruktur für Baum



leerer Baum:



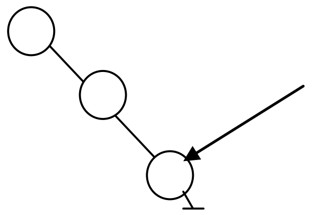
Algorithmus für Search(x):

1. p← root;
2. WHILE (p ≠ NULL && p→key ≠ x) DO
3.   IF x<p→key THEN
4.   p←p→left;
5.   ELSE
6.   p←p→right;
7.   FI;
8. OD

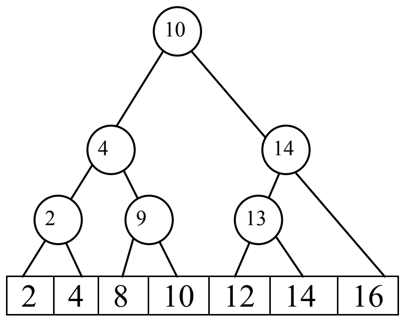
9. return p;

Laufzeit: O(Höhe(T)) für balanicierten Baum T→O(log n)

Variante von Search (für die Insert – Operation) gibt bei erfolgloser Suche nun den zuletzt besuchten Knoten aus

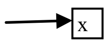


2) Blatt-orientiere binäre Suchbäume
Die Schlüssel (evtl. mit ihren Informationen) werden in den Blätter eines Binären Baumes gespeichert, mit
a) T Hat n-Blätter (n=Anzahl Schlüssel)
b) Die Blätter enthalten von linksnach rechts die aufsteigend sortierte Folge der Schlüssel
c) Die inneren Knoten enthalten Wegweiser, genauer falls Knoten v den Schlüssel i enthält, dann gilt
1) alle Blätter des linken Unterbaumes von x speichern Schlüssel x, mit x ≤ i
2) alle Blätter des rechten Unterbaumes von x speichern Schlüssel x, mit x ≥ i
Beispiel: Schlüssel S={2, 4, 8, 10, 12, 14, 16}



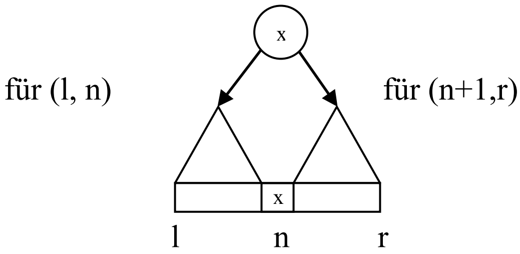
Bemerkung:
1. Man nimmt meistens den max. Schlüssel im linken Unterbaum als Wegweiser
2. n -1 innere Knoten Grund: Alle Schlüssel bis auf den rechtesten kommen als Wegweiser vor
Vorteil: Einige Operationen sind einfacher, Verkettung der Blätter möglich
Nachteil: Doppelter Speicherplatz

Aufbau eines blattorientierten binären Baumes aus sortiertem Feld
Rekursion:
i) Verankerung: 1 Element x (n=1)



1Blatt

ii) rekursiver Aufruf



Höhe: log n

Suche auf einem Blatt-orientierten binären Baum

1. Search(x)
2. p←root;
3. WHILE (p ist kein Blatt) DO
4. IF (x ≤ p→key) THEN
5. p←p→left;
6. ELSE
7. p←p→right;
8. FI
9. OD;
10. return (p→key=x) ? p=Null;

Bemerkung: Nach Termination der while-Schleife in Zeile 9 gilt p→key ist minimal mit p→key ≥ x. Dieses Blatt brauchen wir nur beim Einfügen

Laufzeit: O(Hohe(T)) ist falls T balanciert O(log n)

Update- Operation: Insert & Delete

a) Knoten-orientiert (s=Menge der Schlüssel)

Insert(x) x ∉ S (sonst nichts zu tun)

Search(x) (Variante terminiert in einem Knoten v mit v→key ≠ x } y

Aktionen:

Erzeuge neus Blatt w mit w→key=x

IF x<v→key THEN

v→left←w;

ELSE

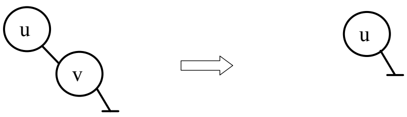
v→right←w;

FI;

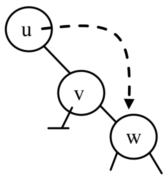
Delete(x) x ∈ S

Search(x) endet in einem Knoten v mit v→key=x

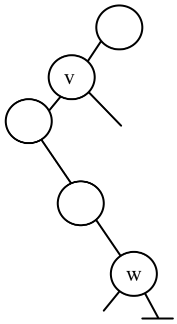
Fall 1: v ist Blatt ➔ entferne v aus Baum



Fall 2: v hat 1 Kind w ersetze v durch w



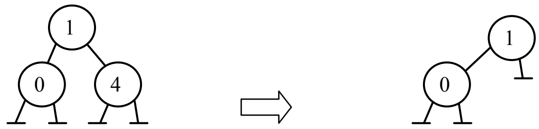
Fall 3: v hat 2 Knoten



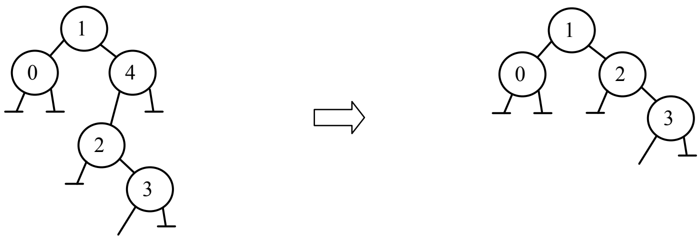
Sei w rechtester Knoten im linken Unterbaum (w→key=y)
Findet man so:

```
P←v→left;
WHILE (p→right ≠ Null) DO
    P←p→right;
OD
```

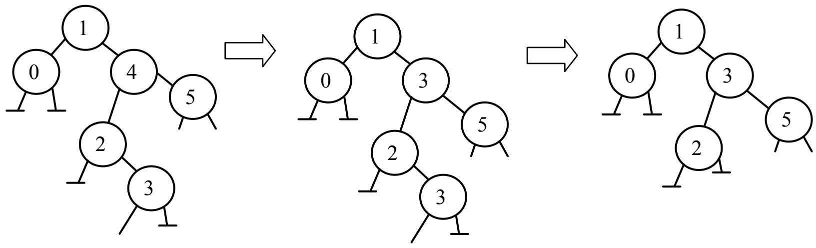
P enthält max Schlüssel y im linken Unterbaum d.h. max. Schlüssel x < x
1) ersetze Schlüssel x durch y v→key←w→key;
2) entferne Knoten w mit w→key=y wie im Fall 1 oder 2 beschrieben (w hat maximal ein Kind)
Beispiele für Delete
Delete(4):
Fall 1:



Fall 2:



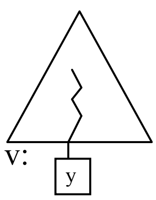
Fall 3:



Blatt-orientierte binäre Suchbäume

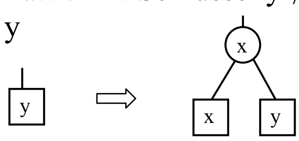
Insert(x) $x \notin S$

Suche nach x Search(x) endet in einem Blatt v mit Schlüssel $y \neq x$

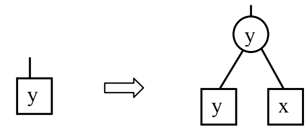


Fall 1: $x < y$

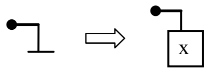
Axiome:



Fall 2: x > y



Sonderfall: Baum vorher leer



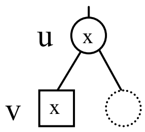
Die Wurzel ist ein Blatt

Delete(x) $x \in S$

Suche nach x endet im Blatt v mit Schlüssel x

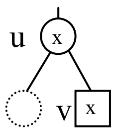
Sei u Vater von v (Sonderfall: u existiert nicht ⇒ v einziges Blatt ⇒ $\perp$)

a) v ist linkes Kind



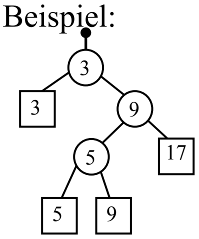
1. Streiche v
2. Ersetze u durch das rechte Kind von u

b) v ist rechtes Kind (symmetrisch)

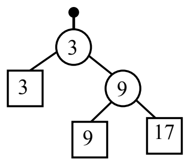


1. Streiche v
2. Ersetze u durch das linke Kind von u

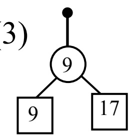
Beispiel:



Delete(5)



Delete(3)



Evtl. muss Wurzel geändert werten root = u

Lemma: Knoten und Blatt-orientierte Suchbäume unterstützen die Wörterbuch Operation, Suchen, Einfügen, Streichen in Zeit O(Höhe des Baums)

Beweis: Suche

Insert / Delete: Suche + Konstante Zahl für Pointer / Schlüsseländerungen

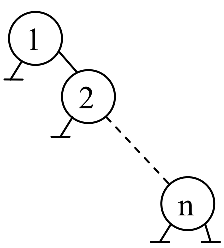
Frage: für einen Suchbaum T: wie groß kann die Höhe werden?

Leider kann die Höhe sehr groß werden → O(n)

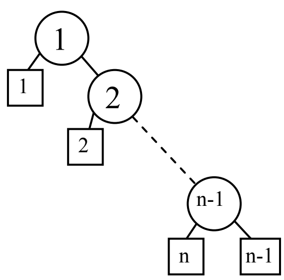
Beispiel für Höhe n (degenerierter Baum)

Füge n einen anfangs leeren Baum die Schlüssel {1, 2, 3, .., n} aufsteigend ein.

Knoten-orientiert



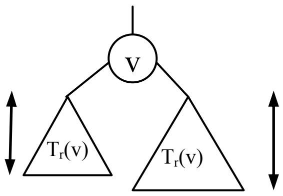
Blatt-orientiert



4.1. Balancierte binäre Suchbäume

(Knoten und Blatt-orientiert)
Garantieren durch lokale Unordnung nach jeder Insert / Delete- Operation eine Höhe von O(log n)
2 wichtige Strategien
Höhen balancierte Bäume
b) Gewichts balancierte Bäume

Höhe von $T_l(v)$



Höhe von $T_r(v)$
(=längster Suchpath)

Gewicht von T_l / T_r = Anzahl der Knoten T_l / T_r

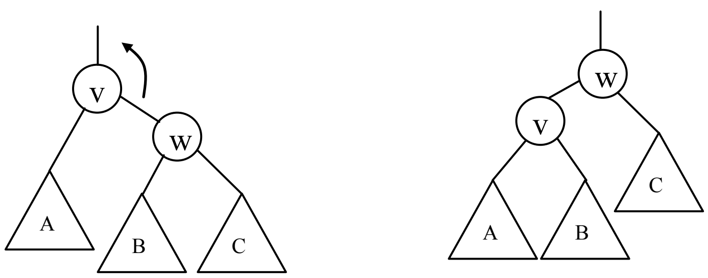
a) Ein Baum T ist höhen balanciert, wenn für alle Knoten v in T gilt:
$|\text{Höhe}(T_l(v)) - \text{Höhe}(T_r(v))| \approx 0$ (nahe bei 0 z.B. ≤ 1)

b) Ein Baum T ist gewichtsbalanciert, wenn für alle Knoten v in T gilt:
$$\frac{\text{Gewicht}(T_l(v))}{\text{Gewicht}(T_r(v))} = 1$$

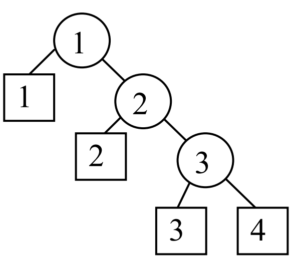
Aus diesen Bedingungen kann man leicht eine O(log n) Schranke für Höhe(T) herleiten

4.1.1. Lokale Umstrukturierung

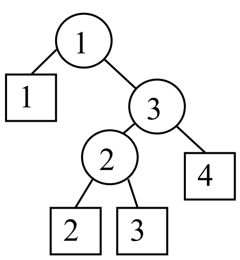
Rotation & Doppelrotation an einem Knoten v (rotate_left(v))
Anwendbar: Wenn Knoten v rechtes Kind hat



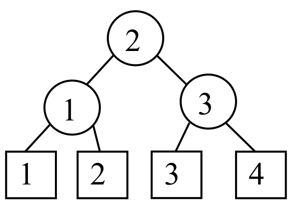
Verletzt nicht Suchstruktur
(Schlüssel-Relationen) des Baumes



Rotation nach links an 2

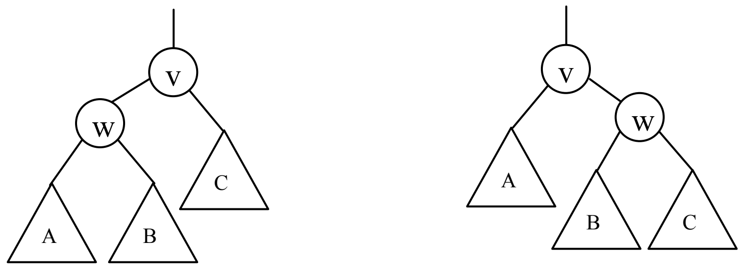


Rotationen nach links an 1

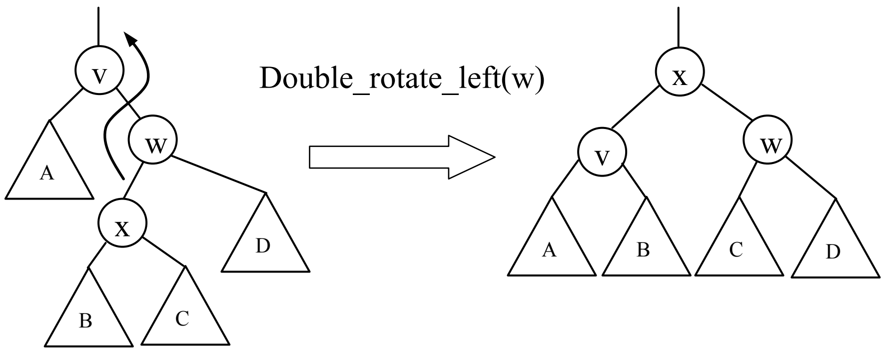


Rotation nach rechts am Knoten v (rotate_right(v))

Anwendung: Wenn Knoten v linkes Kind hat

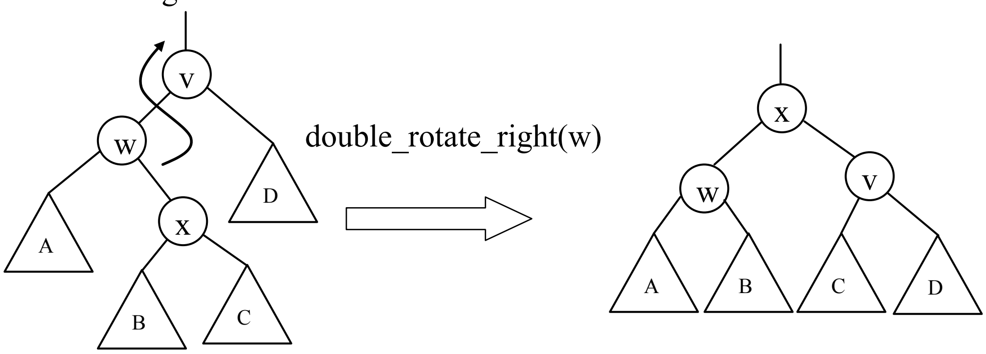


Doppelrotation nach links an Knoten am Knoten v
Anwendbar: Wenn Knoten v ein rechtes Kind w hat und Knoten w ein linkes Kind x hat



=rotate_right(w) + rotate_left(v)

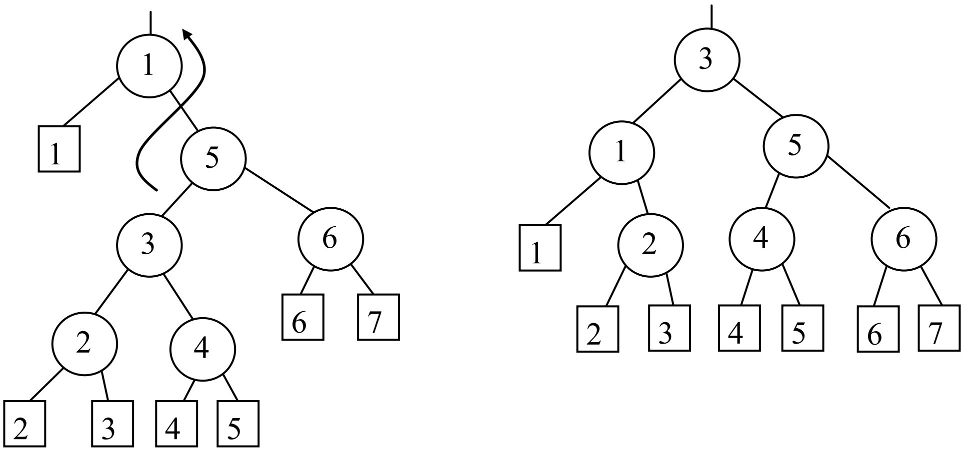
Doppelrotation nach rechts am Knoten v
Anwendung: Falls v linkes Kind w und w rechtes Kind x hat



=rotate_left(w) + rotate_right(v)
Sonderfälle: A, B, C, D leer

- v' ist Wurzel neu Wurzel
  Blatt orientiert!
- Regel gelte innere Knoten Laufzeit O(n)

Beispiel für Doppelrotation

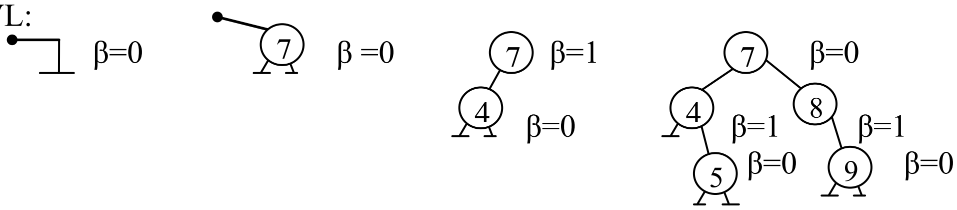


AVL-Bäume (Adelson-Velskii und Landis 1962)

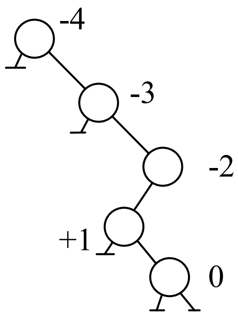
AVL-Bäume sind Höhen balancierte binäre Suchbäume (Knoten-/Blattorientiert)
Definition: Ein AVL-Baum T ist ein binärer Suchbaum, so dass für jeden Knoten v die Höhenstruktur seiner Unterbäume ≤ 1 ist.

$\text{Höhe}(T_l(v))-\text{Höhe}(T_r(v)) \in \{-1;0;1\}:=\beta(v)$

Beispiel: Knotenorientiert
AVL:

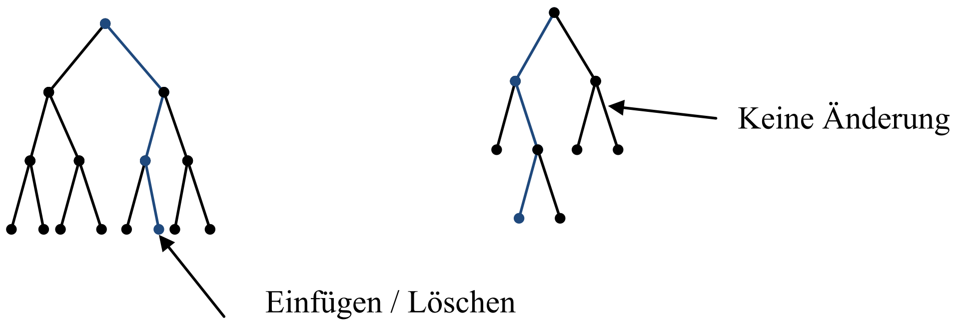


Nicht AVL

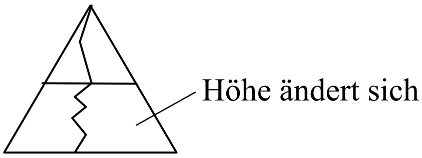


Rebalancierung von AVL-Bäumen

Insert / Delete (vorher gilt AVL-Eigenschaft)
Laufen zunächst einen Suchpfad entlang und fügen einen Knoten hinzu (⇒ Unterbaum wird um eins höher)
Oder nehmen Knoten weg bzw. heben Unterbaum an (⇒ Höhe wird um eins vermindert)
Dies kann dazu führen, das weitere Knoten auf dem Suchpfad ihre Höhe um eins vergrößern oder vermindern



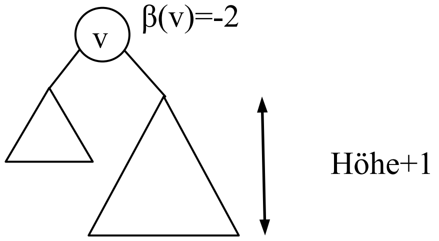
Höhe(v) max Suchpfades der in v startet



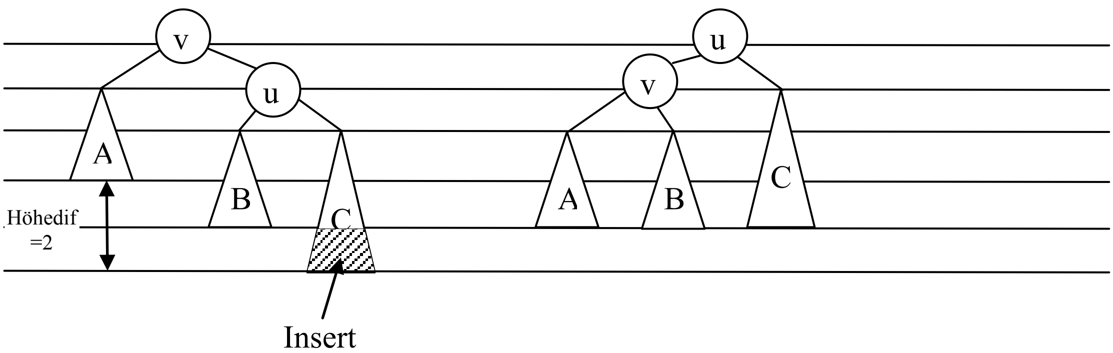
⇒ Dies könnte dazu führen, dass die β-Werte ∉ {-1,0,1}
Rebalancierung

- Laufe Pfad zurück und bringe β durch Rotation in gültigem Bereich
- Durch diese Rotation kann es vorkommen das ein bearbeite Knoten wieder seine alte Höhe erhält (⇒alle Knoten auf Suchpfad haben wieder alte Höhe)
- Dann Stop

Diese Rebalancierung kann gegebenfalls bis zur Wurzel fortsetzen werden
⇒ Kosten Höhe(T) =O(log n)
1) Rebalancierte nach Insert
Insert in einem AVL-Baum vergrößert die Höhe eines Unterbaumes eines Knoten v, so dass $\beta(v) \in \{-2;2\}$
Ohne Beschränkung der Allgemeinheit rechter Unterbaum $T_r(v)$ ist höher

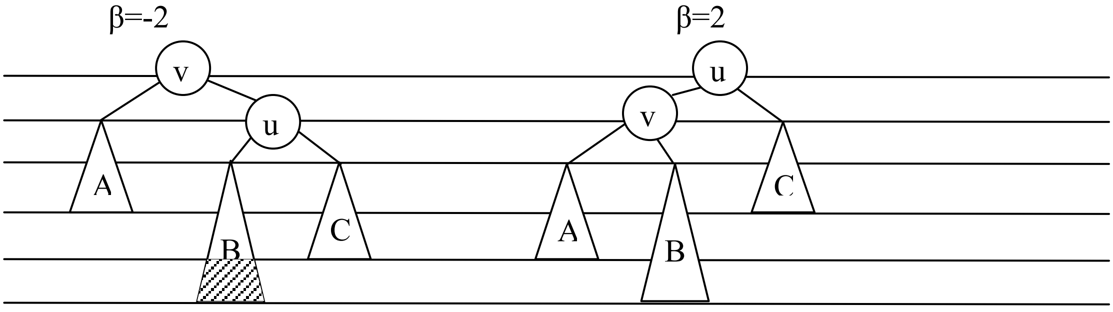


2. Fälle Sei u rechtes Kind von v
Fall1: Insert fand im rechten Unterbaum von u statt:

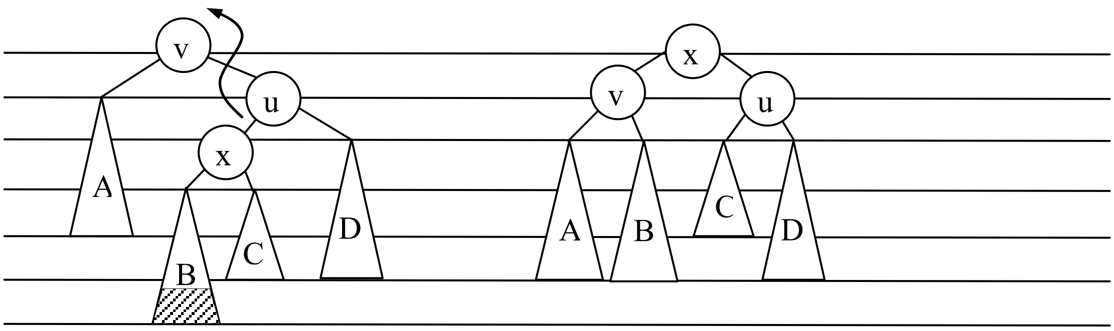


Suchbaum muss genau so aussehen, weil
1) vor dem Insert-Aufruf AVL-Eigenschaft erfüllt war
2) alle Knoten innerhalb von v auch (da bereits balanciert)
⇒ der Suchbaum ist nun AVL-Baum und hat nun die gleiche Höhe wie vor Insert
⇒ Stop

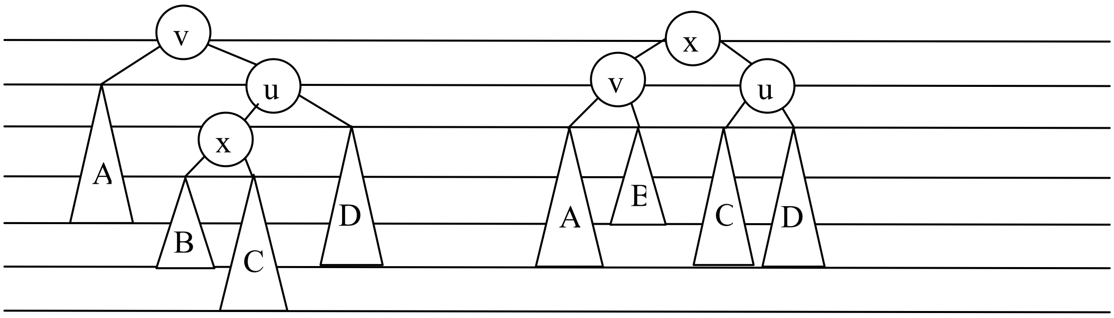
Fall 2: Linker Unterbaum von u ist um eins höher geworden



Genauere Untersuchung von Fall 2 Sei x linkes Kind von u
Fall 2.1



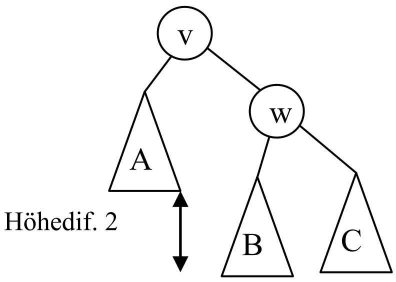
Fall 2.2



⇒ Höhe unverändert
⇒ STOP
weiter 3 symmetrische Fälle β(v) =+2
Insert im linken Unterbaum und v hat die Höhe vom 1 (vergrößert →(Doppel)Rotation nach rechts
Lemma:
Ein AVL-Baum kann auch nach einer Insert-Operation durch eine einzige Rotation bzw. Doppeltrotation rebalanciert werden
Trotzdem: Muss Suchpfad zurück zur Wurzel durchlaufen werden um β-Werte zu korrigieren

2. Rebalancierung nach Delete

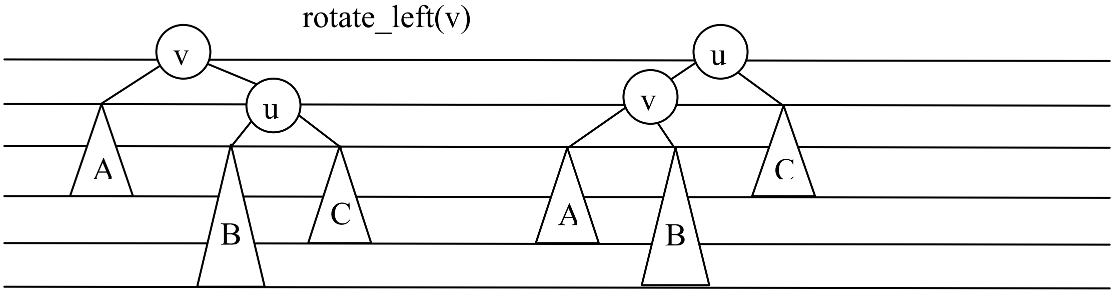
o. b. d. A.: Der linke Unterbaum des Knoten v ist nicht tief genug,
(anderer Fall symmetrisch)
Sei w rechtes Kind von v



Je nach Struktur (Höhe) von B und C verschiedene Fälle

Fall 1:

rotate_left(v)

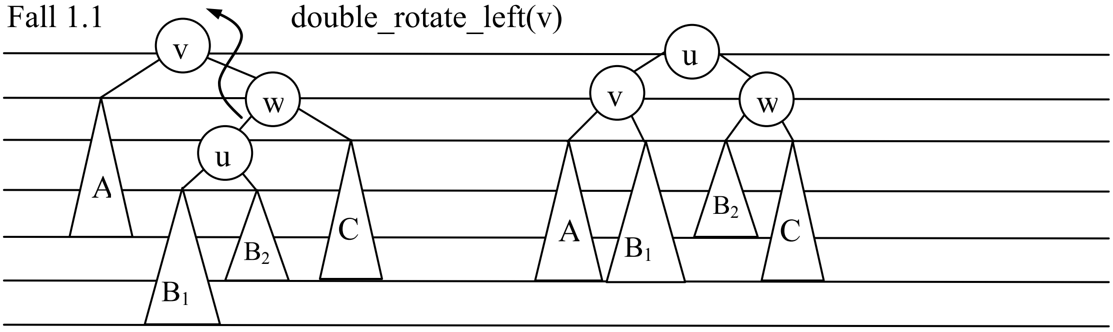


funktioniert nicht

Betrachte die Struktur von Unterbaum B ⇒ 3 Unterbäume

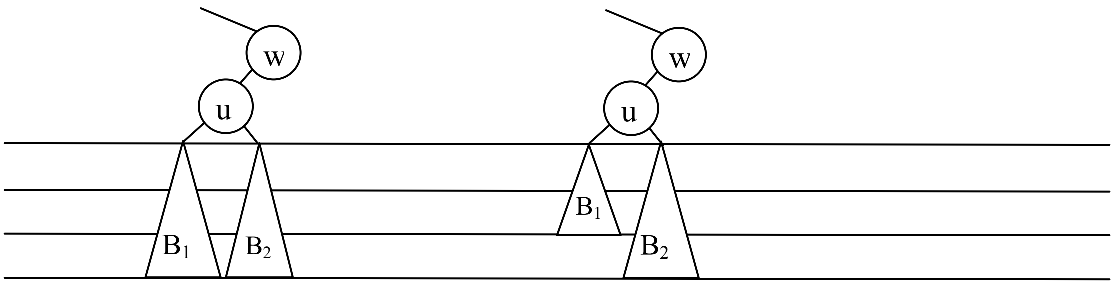
Fall 1.1

double_rotate_left(v)



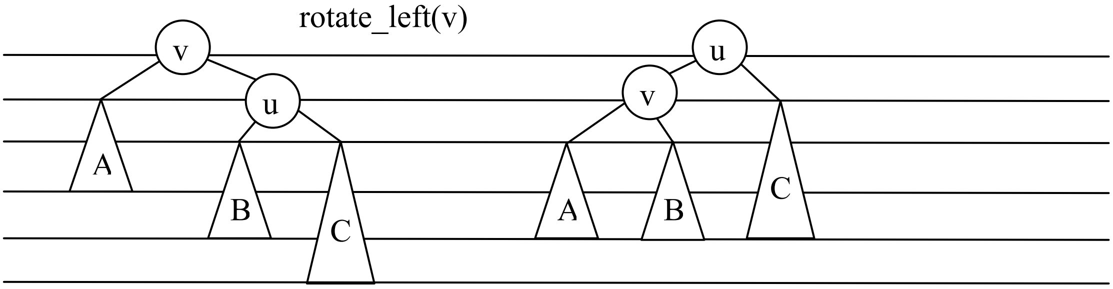
Fall 1.2

Fall 1.3



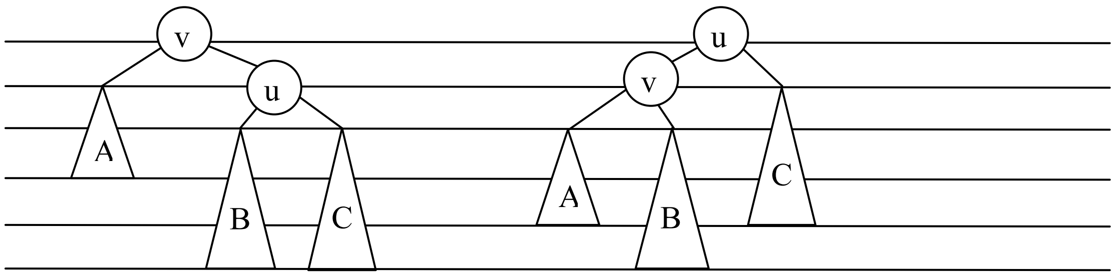
Die Höhe des Unterbaumes insgesamt nach der Doppel_rotation um 1 kleiner als vor der Delete-Operation ⇒ Rebalancierung muss beim Vater von v fortgesetzt werden

Fall 2: C tiefer als B



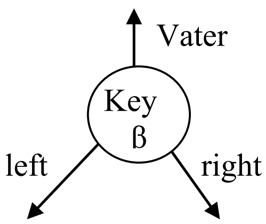
Höhe hast sich um eins vermindert Rebalancierung geht weiter

Fall 3: C und B gleich tief



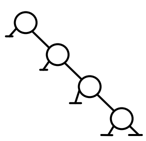
Höhe bleibt hier unverändert wie vor dem Delete ⇒ Stop

Lemma Löschen im AVL-Baum
Ein AVL-Baum kann nach einer Delete-Operation durch eine Folge von (Doppel-)Rotationen entlang des Suchfades zurück zur Wurzel rebalanciert werden
Bemerkung zur Implementierung
1. Zusätzliche Informationen in jedem Knoten ⇒ Höhe bzw. β-Werte
2) Zurück zur Wurzel
   a) Speichern des Pfades bei der Suche in Stack
   b) explizite Vater-Verweise



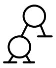
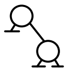
Achtung: Vater-Verweise müssen bei Rotation, Einfügen neu gesetzt werden

Abschätzung der Höhe von AVL-Bäumen
Ziel: Höhe(log n) n=Zahl der Knoten
Idee: Sei N(h) die Minimale Zahl von Knoten in einem AVL-Baum der Höhe n
Zeige, dass N(h) exponetiell mit h wächst
Hinweis im allgemeinen (unbalancierten) Baum ist N(h) = h + 1 möglich



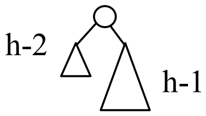
AVL-Baum

N(0)=1 

N(1)=2  bzw. 

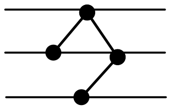
Rekursiv (induktiv) für $n \geq 2$

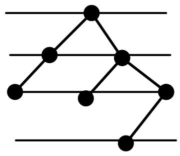
N(h)



$N(h) = 1 + N(h\text{-}2) + N(h\text{-}1)$

Bsp.:

h=2 

h=3 

Ähnlich zu Fibonacci Zahlen

$F_0=0$ $F_1=1$ $F_k=F_{k-2} + F_{k-1}$

Tabelle:

	0	1	2	3	4	5	6	7	8	9	10
F_k	0	1	1	2	3	5	8	13	21	34	55
H(n)	1	2	4	7	12	20	33	54			

Beobachtung $N(h) = F_{n+3} - 1$

Beweis: durch Induktion über h

Induktionsannahme: Tabelle

Induktionsschritt

$N_{h+1} = 1 + N_{h-1} + N_h$

$= 1 + (F_{n+2} -1) - (F_{n+3} - 1)$

$= F_{n+2} - F_{n-3} - 1$

$= F_{n-1} - 1$

Daraus folgt, dass N(h) exponentiell zu h ist

$\Rightarrow$ Höhe(T) = O(log n)

Satz AVL-Baum:

AVL-Bäume unterstützen alle Wörterbücher-Operationen (Look-up, Insert, Delete) in Zeit O(log n) und Platz O(n)

4.2 Spezielle Wörterbücher

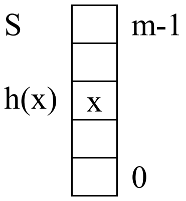
Analog zu speziellen Sortierallgorithmen verwenden die keine (≤)-Vergleiche sondern spezielle Eigenschaften

Hier: Schlüssel sind ganze Zahlen bzw. wir können ihnen ganze Zahlen zuordnen

Hashing

Idee: Sei $S \leq Z$, |S|=m

Speichere S in einer Tafel (Feld) der Größe $m \geq n$ H[0,…,m-1]

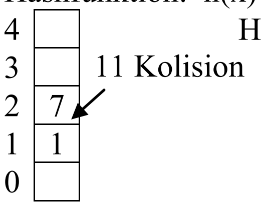


Hash – Funktion

H: Z→{0,…,m-1}

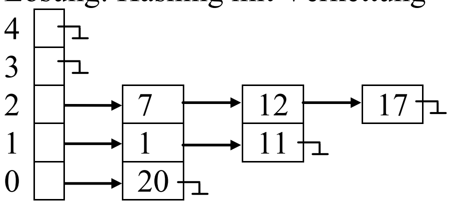
H(x) für x ∈ S gibt Position von x in Tafel

Problem: h muss eigentlich injektiv auf die Menge S sein
Aber S wird durch Insert/Delete verändert
Bsp.: m=S S={1, 7, 11, 12, 17, 20}
Hashfunktion: h(x) = x mod 5



H ist nicht injektiv auf S

Lösung: Hashing mit Verkettung



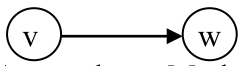
Lookup(x)
i←h(x) // i← x mod 5
Durch Suche Liste H[i] linear nach x

Lookup(12)	Lookup(6)
i←2;	i←6;

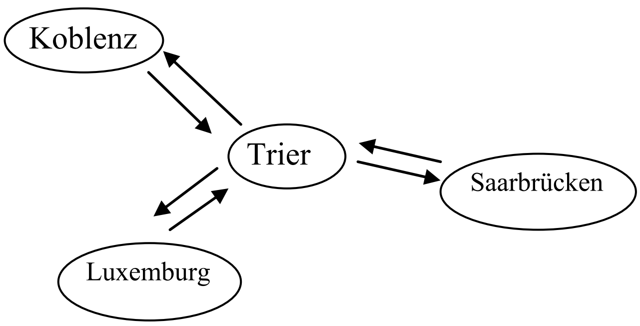
5. Graphen und Graphalgorithmen

5.1 Graph

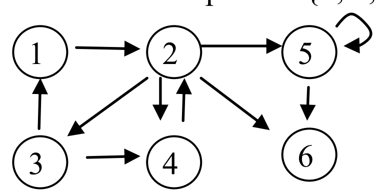
Definition:
Graph G=(V, E) besteht aus einer Menge von Knoten V (verfex) und einer Menge von von Kanten (edge), wobei $E \subseteq V \times V$ (v, w) heißt Kante von v nach w und wird symbolisch dargestellt durch einen



Anwendung: Modellierung komplexer Relationen
Bsp.: Verkehrnetzwerk V=Menge von Städten, E=Bahnverbindungen



andere: chemische Abläufe, soziologische Strukturen
Abstraktes Beispiel: v={1, 2, 3, 4, 5, 6}



E = {(1, 2), (2, 3), (2, 4) , (2, 5) , (2, 6) , (3, 1) , (3, 4) , (4, 2) , (5, 5) , (5, 6)}
Ein Pfad P vom Knoten v zum Knoten w ist eine Folge $V_0,V_1,\ldots,V_L$ von Knoten mit
1. $V_0=v$, $V_L=w$

2. $(V_i, V_{i+1}) \in E$ für i=0..,l-1

l ist die Länge des Pfades (Anzahl der Kanten auf dem Path)

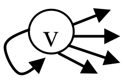
im Bsp. Pfad von 1 nach 6:

1, 2, 5, 6	1, 2, 6	= einfacher Pfad
1, 2, 4, 2,6	1, 2, 4, 2, 4, 2, 4, 2, 6	= 1, 2, $(4, 2)^i$
1, 2, 5, 5, 5, 6		

3. E in Pfad $v_0,\ldots,v_L$ heißt einfach wenn $v_i = v_j$ für alle $i\neq j$ sonst nicht einfach
4. Ein Kreis ist ein Pfad $v_0, v_1,\ldots, v_L$ mit
   1) $L \geq 1$
   2) $V_0 = V_L$

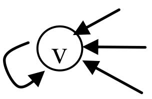
Bsp.: 1, 2, 3, 1 2, 4, 2 5,5

5. Der Ausgangsgrad eines Knotens $v \in V$ outdeg(v)=|w, ∈ V| mit (v, w) ∈ E}|



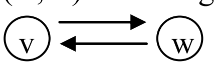
Bsp.: outdeg(4) = 4 outdeg(5) = 2 outdeg(6) = 0

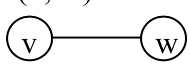
6. Der Eingangsgrad eines Knotens $v \in V$ indeg(v)=|w, ∈ V| mit (w, v) ∈ E}|



Bsp.: indeg(4) = 2 indeg(5) = 2

7. G = (V, E) heißt ungerichtet, wenn gilt $(v, w) \in E \Rightarrow (w,v) \in E$
statt

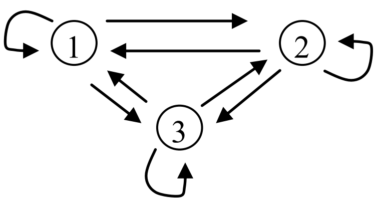




Hier in der Vorlesung
   1) gerichtete Graphen
   2) V = {1, 2,m ,n}
   3) n = |V|, m= |E|

Beachte: $0 \leq m \leq n^2$
leerer vollständiger Graph ($E = V \times V$)

Bsp.: für Vollständigen Graph
n = 3



5.2 Datenstrukturen für Graphen

1) Adjazenzmatrix

Wenn Kanten $(v, w) \in E$, dann heißt w adjazenz zu v (benachtbart) (v, w) heißt auch inzedent zu v

Definition:

Sei G=(V, E) ein gerichter Graph mit n Knoten Die Adjazenz Matrix von G ist eine Boolsche $(n^{x} n)$-Matrix

$A=(a_{ij})$, $1 \leq i, j \leq n$ mit

$a_{ij} = \{ 0, (i,j) \in E; \quad 1, (i, j) \notin E\}$

Bsp.: n = 6

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	1	1	1	1
3	1	0	0	0	0	0
4	0	1	0	0	0	0
5	0	0	0	0	1	1
6	0	0	0	0	0	0

outdeg(i) = Anzahl der 1 in Zeile i

indeg(i) = Anzahl der 1 in Spalte i

Eigenschaften

+ Test, ab Kante $(v, w) \in E$ in Zeit O(1) (nachschauen in Tafel an Position a_{vw})

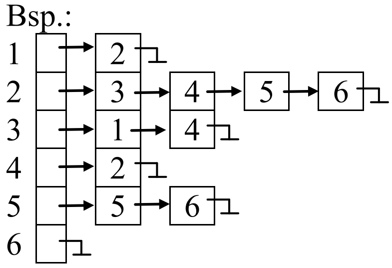
- Platzbedarf $O(n^2)$ Bits

- Iterieren alle Nachbarknoten (d. h. über alle ausgehende Kanten) kostet O(n) → betrachte alle Einträge in Zeile v

2. Adjazenzlisten – Darstellung

Idee: Speichere für jeden Knoten v eine Liste seiner adjazenten Knoten Speichere die Listenanfänge in einem Feld ADJ[1, …, n]

Bsp.:



Eigenschaften

+ Platzbedarf O(n + m) n = Feld ADJ m = Listenelemente

Lineare in Größe des Grafen!

Viel besser als Matrix, wenn $m<<n^2$ z.B. m = O(n) Maximal $O(n^2)$

+ Iteration alle adjazenten Knoten von v (d.h. über die Liste ADJ[v]

Laufzeit: O(1 + outdeg(v)), denn outdeg(v) = Länge der Liste ADJ(v) (optimal!)

- Test, ob $(v, w) \in E$ (d.h. ob v, w benachtbart → Suche linear in ADJ(v) nach w

Laufzeit O(1+ oudeg(v)) → Für die Meisten Graphenalgorithmen nicht erforderlich

Schlussfolgerung:

Wir verwenden Adjazenzlisten

Zur Implementierung:

Objekte für Listenelemente

```
class adj_elem {
int vertex;
adj_elem * next;
// evtl. zusätzliche Daten der entsprechenden Kanten (z.B. Entfernung) }
```

ADJ[1,…, n] speichert adj_elem* Iteration über Nachbarn von v

Pseudo-Code

forall v ∈ V mit (v, w) ∈ E

Implementierung

```
p←ADJ[v]
WHILE p ≠ NULL DO
    w←p→vertex;
    [Rumpf]
    p←p→next;
OD
```

O(1 + outdeg(v)) außer Rumpf

Adjazenz- Darstellung
Platz O(n + m)
Interation über alle ausgehenden Knoten v eines Knotens v in Zeit O(1 + outdeg(v)) Kanten entsprechenden Listenelement (adjunkten) → Daten in Kanten

Interation über alle Kanten (v, w) ∈ E

```
for all v ∈ V   // FOR v=1 to n DO
                // p←ADJ[v];
                // WHILE (p≠NULL) DO
                // w←p→vertex;
                // [Rumpf]
                // p←p→next;
                // od
FOR all w ∈ C mit (v, w) ∈ E DO
[Rumpf]
OD
OD
```

Laufzeit (Ohne Rumpf)

$$O(\sum_{v\in V} 1 + \text{outdeg}(v))$$

$$O(\underbrace{\sum_{v\in V} 1}_{} + \underbrace{\sum_{v\in V} \text{outdeg}(v)}_{})$$

$$=O(\ n \ + \qquad m)$$

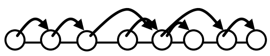
Daten für Knoten v={1,…,n}

Felder A[1,…,n]

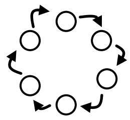
A[v]

1. Problem auf Graphen: Topologisches Sortieren

Definition:
Sei $G = (V, E)$ eingerichter Graph mit $|v| = n$. Eine Abbildung ord: $V \rightarrow [1,.., n]$ heißt topologische Sortierung, wenn gilt:
i) ord ist injektiv
ii) für alle $(v, w) \in E$: $Ord(v) < ord(w)$
Veranschaulichung
Ordne die Knoten auf einem horizontalen Strahl an, so dass alle Kanten von links nach rechts verlaufen.



Beobachtung: Topologische Sortierung existiert nur, wenn G keine Kreise enthält G ist azyklisch
Gegenbeispiele:



notwendige Bedingung
Lemma: Ein Graph G=(V, E) besitzt genau dann eine topologische Sortierung, wenn G azyklisch
top. Sortierung $\Longleftarrow\!\Longrightarrow$ azyklisch
„$\Rightarrow$": folgt aus Beobachtung
„$\Leftarrow$": Induktion über Zahl der Knoten n
n=1 (Induktions Anfang)
V ○ $\Rightarrow E = 0$
sonst Selbstschleife $\Rightarrow$ zyklisch $\Rightarrow \exists$ top. Sortierung ord(v)=1
$n \Rightarrow n + 1$ Induktions Schritt
Induktionsannahme: Für alle azyklische Graphen mit n Knoten existiert top Sortierung Sei G azyklisch mit n + 1 Kanten
Behauptung G besitzt einen Knoten mit Eingangsgrad 0 ($\exists\, v \in V$ mit indeg(v)=G)
Bew. betrachte folgenden Algorithmus
1. Starte in beliebigen Knoten v
2. solange indeg(v)>0 DO
3. sei (u, v) eine eingehende Kante
4. $v \leftarrow u$;
5. OD

Algorithmus terminiert da
i) kein Knoten wird mehrmals besucht, da G azyklisch
ii) Knotenmenge ist endlich
indeg(v) =0 bei Termination
Konstruiere aus G einen Graphen G` mit n Knoten
- wähle Knoten v mit, indeg(v)=0 (existiert nach Behauptung)
- entferne v und seine eingehende Kanten
Es gilt:
1) G' hat n Kanten
2 G' ist azyklisch

Induktions Anfang $\Rightarrow$ G' hat top. Sortierung ord: $V \rightarrow \{1,\ldots,n\}$
Dann besitzt G die folgende top. Sortierung ord: $V \rightarrow \{1,\ldots,n\}$
ord(v)={1, falls $u = v$; ord(v +1), falls $u \neq v$

Auflösen der Induktion bzw. Rekursion führt zu folgender Algorithmischen Idee
1. suche Knoten v mit indeg(v)=0
2. gib v nächste Nummer
3. entferne v + ausgehende Knoten

wiederhole 1-3 bis v=0
Pseudo-Code

```
1. count ←0;
2. WHILE ∃ Knoten mit indeg=0 DO
3. wähle solchen Knoten v;
4. ord[v]←++count;
5. G ←G\{v}; OD
6.  IF count <n then
7. ERROR: "G ist zyklisch"
8. FI
```

2 Problem
1. Wie findet man in Zeile 2 bzw. 3 effizient einen Knoten mit indeg =0?
2. Algorithmus sollte G nicht zerstören (durch entfernen von Knoten (Zeile 5)

Lösung
1. Verwalte eine Kandidatenliste ZERO der Knoten mit Eingangsgrad 0

i) Initialisierung Finde die Knoten mit indeg=0
ii) in Zeile 5 müssen evtl. Nachbar von v aufgenommen werden

2. In Zeile 5 lösche keine Knoten Simuliere löschen durch Verwaltung (Speicherung) der entsprechenden Eingangssprache in einem separaten Feld INDEV d.h. INDEV(v)=indeg(v), wenn lösch-Operationen ausgeführt werden

$\Rightarrow$ InZeile 5
Vermindere INDEV[w] für alle Nachbarn w von v um 1. wenn dann INDEG[w]=0 füge w i Liste ZERO ein
Dies führt zu folgendem Algorithmus (2. Version)

```
1. count←0;
2. FORALL v ∈ V do INDEG[v]←0; OD;
3. FORALL (v, w) ∈ E DO INDEG[w]++; OD;
4. ZERO←Ø;
5. FORALL (v, w) ∈ E DO
6.      IF INDEG[v]=0 then
7.           ZERO←ZERO U {v}
8.      FI
9. OD
10. WHILE ZERO ≠ Ø DO
11.     v←ZERO \[v};
13.     ord[v]←++count;
14.     FORALL w ∈ V mit (v, w) ∈ E DO
15.          IF (--INDEG[v]=0) THEN
16.               ZERO←ZERO U {w}
```

```
            FI
      OD
OD
IF count < n THEN
      ERROR: “G ist zyklisch”
FI
```

Realisierung der Menge S besuchte Knoten)

Gute Realisierung Bitvektor (Boolesches Array) besucht[v] = true $\Longleftrightarrow$ v $\in$ S Menge $\tilde{S}$ später

3. Verfeinerung des Algorithmus

Explorefrom(s)
besuchte[s] $\leftarrow$ true; //S $\leftarrow$ S U {s}
$\tilde{S} \leftarrow \{s\}$
while $\tilde{S} \neq \emptyset$ do
    v $\leftarrow$ beliebiger Knoten aus $\tilde{S}$
    if P[v] = Null //Kein unbenutzte Knoten aus v
    then $\tilde{S} \leftarrow S \setminus \{u\}$;
    else w $\leftarrow$ P[v]$\rightarrow$vertex;
        P[v]$\leftarrow$P[v]$\rightarrow$next; //Markiere als benutzt
        if $\neg$ besucht[w] //neuer Knoten
        then besucht[w] $\leftarrow$ true;
            $\tilde{S} \leftarrow \tilde{S}$ U {w}
        fi
    fi
od

Für alle von S aus erreichbaren Knoten gilt nach Termination besucht[v] = True
Für alle nicht erreichbar besucht[v] = false $\Rightarrow$ Ausgabe
Andere Ausgabe-Variante immer, wenn besucht[v] gesetzt wird
- hänge entsprechenten Knoten an Liste
- textuelle Ausgabe (Datei)

Initialisierung

forall v $\in$ V do
    besucht[v] $\leftarrow$ false;
P[v]$\leftarrow$ADJ[v];
od

Wie realisiert man die Menge $\tilde{S}$

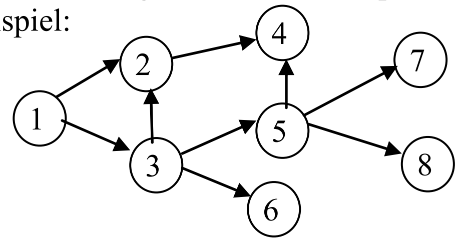
Operation:
1) $\tilde{S} \leftarrow \emptyset$
2) $\tilde{S} \leftarrow \tilde{S}$ U {v}

$\left.\begin{array}{l} 1) \\ 2) \end{array}\right\} \tilde{S} \leftarrow \{v\}$

3) Auswahl und entfernen eines beliebigen Element
4) Test auf leere Menge

Zwei Mögliche Datenstrukturen
a) Keller (Stack)
b) Schlange (queue) $\Rightarrow$ Laufzeit für alle Operationen O(1)

Ablauf des Algorithmus am Beispiel einmal mit Stack und einmal queue für $\tilde{S}$
Beispiel:



Explorefrom(1)

$\tilde{S}$ als Stack: DFS:

$\tilde{S}$	S (besucht)
1	1
21	12
421	124
21	124
1	124
31	1243
531	12435
7531	124357
531	124357
8531	1243578
531	1243578
31	1243578
631	12435786
1	12435786
\	12435786

$\tilde{S}$ als Queue BFS

$\tilde{S}$	S (besucht)
1	1
12	12
123	123
23	123
234	1234
34	1234
345	12345
3456	123456
456	123456
56	123456
567	1234567
5678	12345678
8	12354678
\	12345678

Laufzeit hängt von der Größe von s aus erreichbaren Teilgraphen
Genauer:
Definition: Sei G=(V, E) eingerichter Graph und V' $\subseteq$V Der Graph $\subseteq$ (V', E) mit
$E' = E \cap (V' x V')$

d.h. E' = {(v,w) $\in$ E] v, w $\in$ V' der von V' induzierte Teilgraph von G
Lemma:
Sei S die durch einen Aufruf explorefrom(s) besuchte Knotenmenge und sei u_s=|S| und m_s = Anzahl der Kanten des von S induziertes Teilgraphen. Dann hat explorerfrom(s) die Laufzeit $O(u_s + m_s)$
Beweis: Der Rumpf der while-Schleife kostet O(1) In jeder Ausführung des Rumpfes wird entweder ein Knoten von S genau einmal in $\tilde{S}$ aufgenommen wird. (Test Zeile 10) und jede Kante höchstens einmal benutzt wird (P[v]-Zeiger), wird die While Schleife höchstens (n_s + m_s) mal ausgeführt $\Rightarrow$
$O(n_s + m_s)$
Achtung: Initialisierung O(n + m)

Eine genauere Untersuchung von DFS (Tiefensuche)
1. Eine rekursive Version von DFS
Eingabe Graph G = (V, E)

Datenstruktur: Bitvektor: besucht
Initialisierung:

forall $v \in V$ do besucht[v]← false; do

Rekursive Funktion alle von v aus erreichbaren Knoten in DFS-Reihenfolge

1. dfs(v)
2. besucht[v]←true;
3. forall $w \in V$mit $(v, w) \in E$ do
4. if $\neg$ besucht[w] then dfs[w] fi
5. od

Ablauf von dfs(1) am Bespiel:

$$\begin{tikzcd}[column sep=small, row sep=tiny]
 & \text{dfs}(3) \arrow[r] \arrow[drr] & \text{dfs}(6) & & \text{dfs}(4) \\
\text{dfs}(1) \arrow[ur] \arrow[dr] & & & \text{dfs}(5) \arrow[ur] \arrow[r] \arrow[dr] & \text{dfs}(7) \\
 & \text{dfs}(2) \arrow[r] & \text{dfs}(4) & & \text{dfs}(8)
\end{tikzcd}$$

besucht: 1, 2, 4,3, 5, 7, 8, 6
verschachtelte dfs-Aufrufe
entsprechenden Stack
Genauere Untersuchung der Zeile 4
Wir Klassifizieren die benutzte Kante in 4 Mengen (Klassen) T, F, B, C Sei (v, w) die aktuelle Kante in Zeile 4
Baum Kanten T tree

(v, w) heißt Baumkante d.h. $(v, w) \in T$, wenn w nicht besucht

1. dfs(v)
2. besucht[v]←true;
3. forall $w \in V$ mit $(v, w) \in E$ do
4. if $\neg$ besucht[w] then
5. dfs(w);
6. T←T U {(v, w)}
7. fi
8. od

Einteilung der Kanten in 4 Klassen T, F, B, C

1. Baumkanten T

Kanten (v, w), die in Zeile 4 zu noch nicht besuchten Knoten w führen
Baumpfade: Pfade aus Baumkanten

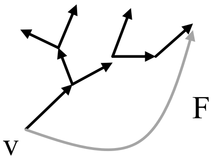
$v \overset{K}{\underset{T}{\to}} w$

2. Vorwärtskanten F (Forward)

$(v, w) \in F$, falls (in Zeile 4) w schon besucht und $v \overset{*}{\underset{T}{\to}} w$ es existiert ein Baumpfad der Länge

≥ 1 von v nach w

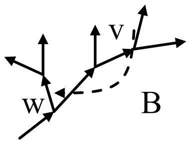
$\to$T
$\to$F

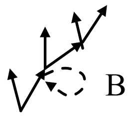


3. Rückwärtskanten B (Backward)

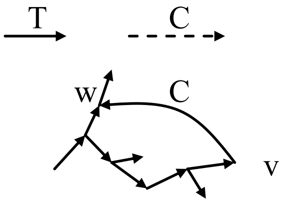
$(v, w) \in B$, wenn (in zeile 4) w schon besucht und $w \overset{K}{\underset{T}{\to}} v$

Baumpfad von w nach v (w = v möglich)





4. Querkanten (Cross-Kante)
alle übrigen Kanten, d.h. w schon besucht und ∄ Baumpfad zwischen v und w



Wir erweitern den dfs-Algorithmus
1) Berechne die Klassen T, F, B, C (Konzeptuell)
2) Berechne 2 Nummerierung der Knoten:
dfsnum: Reihenfolge der rekursiven dfs-Aufrufe
compnum (conplitien): Reihenfolge in der dfs-Aufrufe abgeschlossen werdem

Hauptprogramm:

forall $v \in V$ do
    besucht[v] ← false;
od
$count_1$ ← 0;
$count_2$ ← 0;
[T, B, C, F ← Ø]

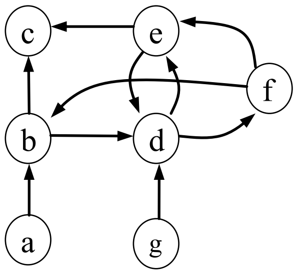
} Init.

forall $v \in V$ do
    if ¬ besucht[v] then dfs(v)
od

dfs(v)
besucht[v] ← true;
dfsnum[v] ← $count_1$;
forall $w \in V$ mit $(v, w) \in E$ do
    if ¬ besucht[w] then
        [T ← T U {(v, w)}
        dfs(w);
    else if ($v \xrightarrow[T]{*} w$) then
        F ← F U {(v, w)}
    else if ($w \xrightarrow[T]{K} v$) then B ← B U {(v, w)}
        else C ← C U {(v, w)}

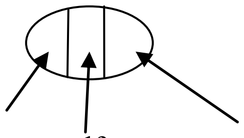
```
            fi
        fi
    fi
od
compnum[v] ← ++count₂;
```

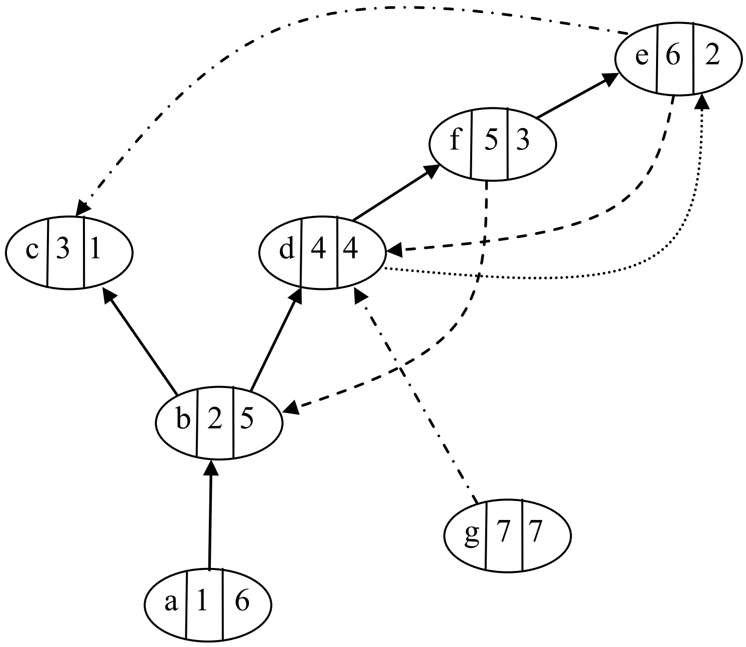
Hauptprogramm realisiert einen DFS-Lauf auf dem Graphen
Beispiel:



T —— als Baum gezeichnet
F ⋯⋯ von oben nach unten
B - - - -
C -·-·-

v = {a, b, c, d, e, f, g}





Wichtig: DFS-Struktur (Einteilung T,F, B, C) hängt ab von konkreter Datenstruktur des Graphen! Insbesonders Reihenfolge der Knoten und Adjazenzlisten

Hauptprogramm:
forall $v \in V$ do
    besucht[v]$\leftarrow$false;
od
count$_1 \leftarrow 0$;
count$_2 \leftarrow 0$;
[T, B, C, F$\leftarrow \emptyset$]

} Init

forall $v \in V$ do
    if $\neg$ besucht[v] then dfs(v) fi
od

dfs(v)
besucht[v]$\leftarrow$true;
dfsnum[v]$\leftarrow$++count$_1$;
forall $w \in V$ mit $(v, w) \in E$ do
    if $\neg$ besucht[w]
    then [t$\leftarrow$T U {(v, w)}
        dfs(w)
    else if ($v \xrightarrow[T]{*} w$)
        then F$\leftarrow$F U {(v, w)}
        else if ($w \xrightarrow[T]{K} v$)then B$\leftarrow$B U {(v, w)}
            else C$\leftarrow$C U {(v, w)}
            fi
        fi
    fi
od
compnum[v]$\leftarrow$++count$_2$;

Effiziente Berechnung des Klassen T, F, B, C mit Hilfe der Nummerierung: dfsnum, compnum

Lemma:

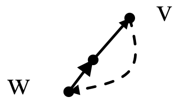
a) T, F, B, C ist Partition von E

b) T entspricht dem Aufrufbaum der (rek.) dfs-Aufrufe

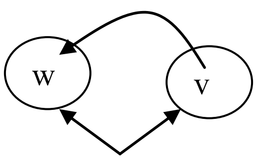
c) $v \xrightarrow[T]{+} w \iff$ dfsnum[v] $\leq$ dfsnum[w] ^ compnum[v] $\geq$ compnum[w]

d) $\forall$ $(v, w) \in E$: $(v, w) \in E$: $(v, w) \in T \cup F \iff$ dfsnum[v] < dfsnum[w]

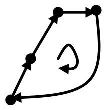
e) $\forall$ $(v, w) \in E$: $(v, w) \in B \iff$ dfsnum[w] $\leq$ dfsnum[v] ^ compnum[w] $\geq$ compnum[v]



f) V $(v,w) \in E$ $(v, w) \in C \iff$ dfsnum[w] < dfsnum[v] ^ compnum[w] < compnum[v]

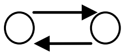


Folgerung (aus Darstellung)
1. Partitionierung der Kanten in T, F, B, C kann effizient durch Anwendung der entsprechenden Teile des Lemma nach einer DFS-Lauf
a) Nummerierung (divide & conquer) wie die Einteilung in T, F, B, C ist nicht eindeutig, sondern hängt von der Reihenfolge der Knoten und Kanten in der ADJ-Liste ab.
b) In azyklischen Graphen gilt immer B=Ø d.h. es werden keine Rückwärtskanten produziert.



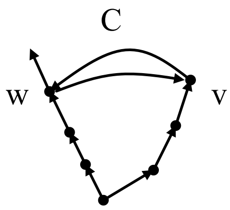
Lemma $\Rightarrow$ Teile e) $\quad \forall\, (v, w) \in E$ compnom(v) > compnom(w)

$\Rightarrow$ Man kann aus compnom eine Topologische Sortierung berechnen
DFS-Lauf auf einem ungerichteten Graph produziert keine Crosskanten (C=Ø) ungerichter Graph G=(V, E) $\quad \forall\, (v,w) \in E \quad (w, v) \in E$



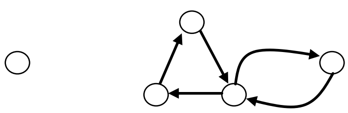
v w

v w

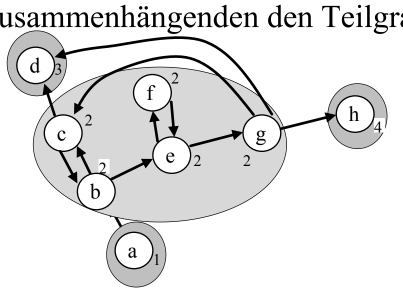


Begründung: Der DFS-Aufruf von w hätte den Knoten v mitbesucht (wegen Gegenkante)

Eine Anwendung von DFS
Die Berechnung des Starkenzusammenhang Komponeten eines gerichteten Graphen
Definition:
a) Ein gerichteter Graph G=(V, E) der heißt stark zusammenhängend, wenn gilt $\forall\, (v, w) \in V$
v$\bullet\!\!\longrightarrow\!\!\bullet$w es existiert ein Path von v nach w
Beispiel:

starkzusammenhängend:	Nicht stark zusammenhängend
	

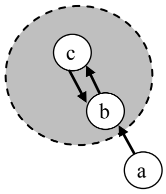
b) Die Starken zusammenhängenden Komponeten ($S \supseteq K$) von G. Sind die maximal stark zusammenhängenden den Teilgraph von G

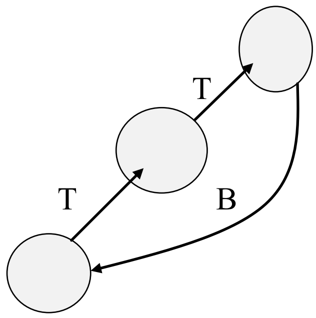


$K_1 = \{a\}$
$K_2 = \{b, c, e, f, g\}$
$K_3 = \{d\}$
$K_4 = \{h\}$

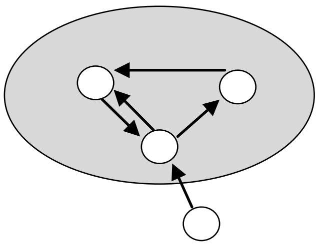
Alternative Darstellung Komponeten Maximal SZK-num V→{1,…,n} (wobei K=# der SZK)
mit SZK_num(v) = SZK_num(w) $\Longleftrightarrow$ v und w liegen im selben SZK
Idee für Algorihtmus:
füren einen DFS-Lauf aus. Sei G'=(V', E) der Teilgraph von G aufgespannt von der bereits besuchten Knoten V' und den besuchten Kanten E' (d.h. G' ist der Teilgraph den DFS kennt)
Initialisierung
V'←Ø;
E'←Ø;
SZK←Ø; //SZK=Menge von Mengen

Ablauf am Beispiel:
1. besuchter Knoten a → V'={a}; E'=Ø; SEK={{a}}
2. Sei (v,w) nächste betrachtete Kante (am Beispiel (a,b))
Fall 1: (v,w) $\in$ T (d.h. w nicht besucht → Füge neue Komponete {w} zu SZK hinzu V'←V' U {w} E'←E' U {(v, w)}
Am Beispiel: V'={a, b} E'={(a, b)} SZK={{a}, {b}}
Fall2 nächste betrachtete Kante (v, w) $\notin$ T (v, w) können mehre Komponeten in SZK zu einer Komponete zusammen mischen
Fall 2.1 (v, w) $\in$ B





SZK={{a},{b,c}}
Fall 2.1
(v, w) $\in$ C

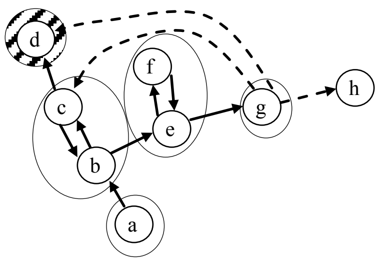


Fall 2.3 (v, w) $\in$ F
Kann ignoriert werden da kein neuer Path angezeigt (wir kenne schon den Baumpfad v $\xrightarrow{T}$ w

Problem: Effizientes Mischen der Komponenten (Fall 2.1 und 2.2)
einige Definitionen und Bezeichnungen
a) Eine Komponente heißt abgeschlossen, wenn die DFS-Aufrufe von dfs(v) für alle v $\in$ K abgeschlossen ist
b) Die Wurzel einer Komponeten K ist der Knoten in K mit der kleinsten dfs_num (wurde als erstes benutzt)
c) unfertig = Folge der Knoten für die dfs aufgerufenen wurde, aber deren Komponeten noch nicht abgeschlossen ist, nach dfsnum sortiert

d) wurzeln = Folge der Wurzeln der noch nicht abgeschlossenen Komponeten, naach dfsnum sortiert
Im Beispiel: Situation, wenn DFS Knoten g erreicht hat

abgeschlossen
gesehen
ungesehen



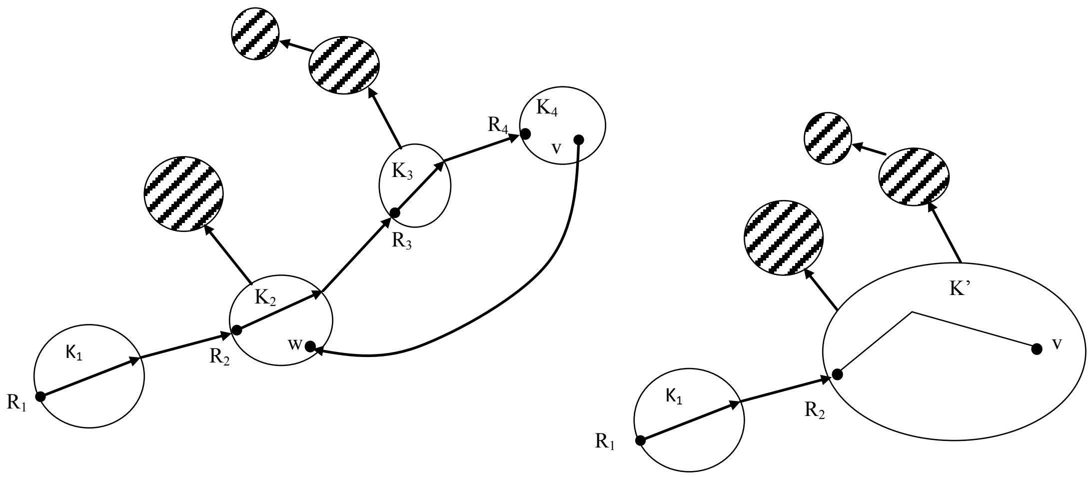
unfertig: a, b, c, e, f, g
wurzeln: a, b , e , g

nicht abgeschlossene Komponenten

Beobachtungen:
i) $v \in$ unfertig $\Longleftrightarrow v \xrightarrow{*} g$ (g= aktueller Aufruf)
ii) $\nexists\ (v, w) \in E$ mit v in abgeschlossener und w in nicht abgeschlossener Komponente
Wir betrachten nun die Kanten aus g heraus dfs(g)
$(g, d) \in C$ es passiert nicht, da d in bereits abgeschlossener Komponente, $\Rightarrow$(g, d) kann kein Kreis schließen! $(g, c) \in C$ vereingt die 3 Komponenten mit den Wurzeln b, e, g
$\Rightarrow$ durch Streichen der Wurzeln e und g aus wurzeln
unfertig: a, b, c, e, f, g
wurzeln: a, b

2 nicht abgeschlossene Komponeten
$(g, h) \in T$ neue Komponete {h]}
unfertig: a, b, c, e, f, g, h fügen h zu unfertig und wurzeln hinzu
wurzeln: a, b, h
Bei Rückkehr aus einen Aufruf dfs(v) (dort wo die completion-Nummer vergeben wird) testen wir, ob v eine Wurzel ist (v = rechtes Elemente von wurzeln?)
Falls ja, dann geben wir die Komponente mit Wurzel v aus (die rechteste Komponente!) und entfernen sie und ihre Wurzel aus unfertig bzw. wurzeln
Beachte das hinzufügen und entfernen von Knoten geschieht immer am rechten Ende der beiden folgen $\Rightarrow$ Stack

Allgemeine Situation:

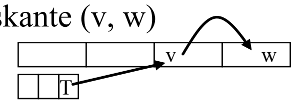


	K_1				K_2				K_3				K_4				
Unfertig:	R_1				R_2				R_3				R_4				(v, w) ∈ B U C
Wurzeln:	R_1				R_2				R_3				R_4				

Aktion: While (dfsnum[wurzel.top()] > dfsnum[w]) do
wurzeln.pop();
od;

Vorwärtskante (v, w)

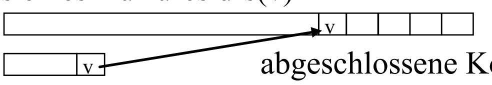
unfertig
wurzel



es passiert nix dfsnum[v]≤ dfsnum[w]

Abschluss eines Aufrufes dfs(v)

unfertig
wurzeln



abgeschlossene Komponente

Aktion (am Ende von dfs)
if (v = wurzel.top()) then
repeat w←unfertig.pop();
gib den Knoten w aus
until w= v
wurzel.pop();
fi

Komplettes Programm
Ausgabe: SZK_count = Anzahl an SZK
Feld SZK_num
AZK_num[v] = SZK_num[w]
⟺ v und w in der selben SZK
globale Daten
Zähler: dfs_count SZK_count
Felder: besucht[v] dfsnum[v] SZK_num[v]
Stacks: unfertig wurzeln
Bitvektor: inunfertig[v] = True ⟺ v ∈ unfertig

Hauptprogramm:

```
1. dfs_count ←Ø;
2. SZK_count←Ø;
3. unfertig.clear();
4. wurzeln.clear();
5. forall v ∈ V do
6.      besucht[v] ←false
7.      inunfertig[v]←false;
8. od
9. forall v ∈ V do
10. if ┐ besucht[v] then
11.     dfs[v]
12. fi
13. od
```

Die erweiterte dfs-Funktion

```
1. dfs(v)
2. dfs_count++;
3. dfsnum[v]←dfs_count;
4. besucht[v]←true;
6. wurzeln.push(v)
7. inunfertig[v]←true;
8. forall w ∈ V mit (v, w) ∈ E do
9. if ┐ besucht[w]
10. then dfs[w]          // (v, w) ∈ T
11. else                 // (v, w) ∉ T
12.     if infertig(v) then     //mischen
13.          while dfs_num[wurzel.top()]> dfs_num[w] do
14.               wurzeln.pop()
15.          od
16.     fi
17. fi
18. if v = wurzel.top() then    //neue abgeschlossene Komponete SZK
19.     SZK_count++;
20. repeat w←unfertig.pop();
21.     inunfertig(w)←false;
22. SZK_num(w)←SZK_count;
23. until w = v;
24.     wurzeln.pop();
25. od
```

Laufzeit

i) reine dfs-Laufzeit

- im Rumpf von dfs(v) ohne weitere rekursive Aufrufe ist die Laufzeit O(1 + outdeg(v))
- dfs(v) wird genau einmal für jeden Knoten v aufgerufen ⇒ Laufzeit

$$O\sum_{v\in V} 1+\text{outdeg}(v)=O(n+m)$$

ii) zusätzliche Aufwand für

- Verwalten der Keller
- Berechnen der SZK_num

ist O(1) pro Knoten

Denn Jeder Knoten ist genau einmal im unfertig-Keller und höchstens einmal im wurzeln-Keller

⇒ Satz: Die starken zusammenhangenden Komponenten eines gerichten Graphen können durch einen DFS-Lauf in Zeit $O(n + m)$ berechnet werden