5 Datenstrukturen: Listen, Bäume, Graphen

Lernziele:

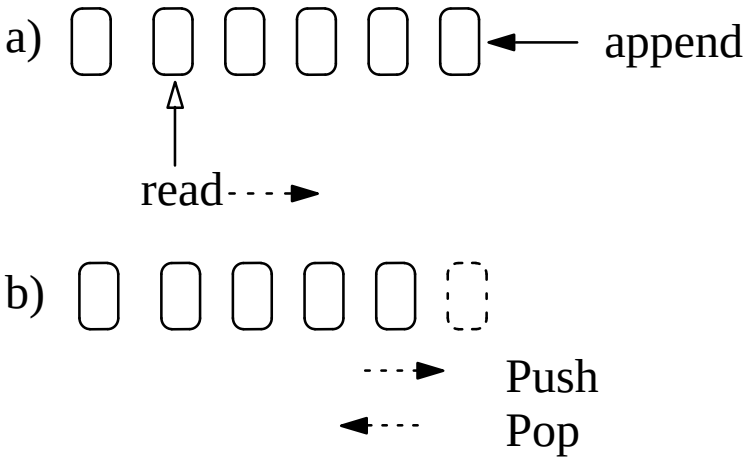
- Kennenlernen von Standard-Datenstrukturen
- Datenstrukturen als Anwendung von Datenabstraktion
  - Realisierung hängt von Schnittstelle ab
  - Realisierungsvarianten für eine Schnittstelle
- Algorithmen, die Datenabstraktionsbausteine verwenden
- verschiedene Tradeoffs, z.B. Laufzeit und Speicherplatz

Lineare Listen

Lineare Listen mit verschiedenen Zugriffsmöglichkeiten

– im folgenden ist Elementtyp ein Text, in C vom Typ `char*`

– wir betrachten im folgenden Abschnitt 2 Listenarten



Liste hinten beschrieben von vorne gelesen (Schlange, FiFo Datenstruktur)

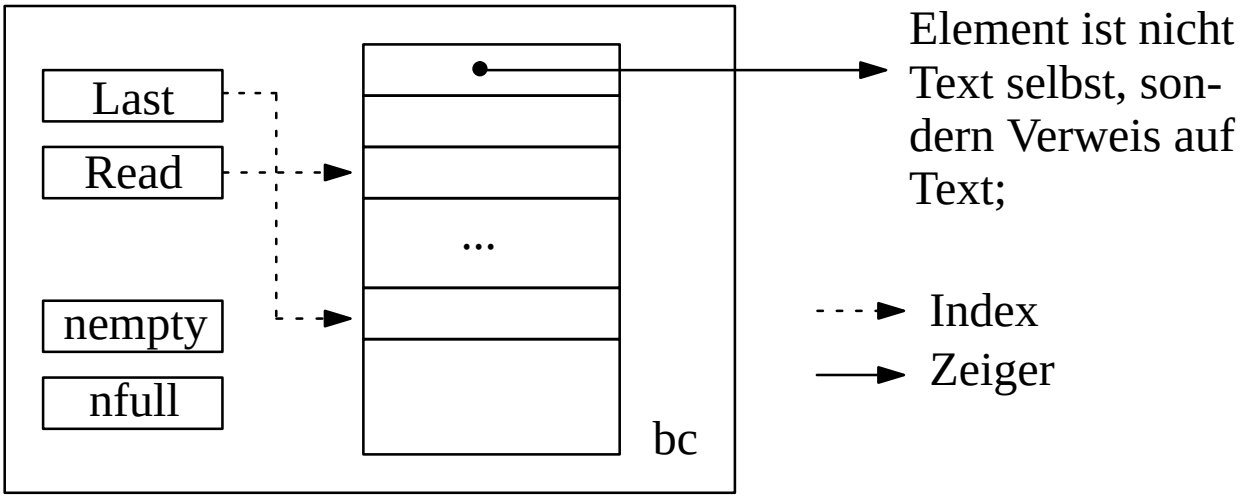
Keller (stack): LiFo Datenstruktur

– Schnittstelle der Append–Liste als ADT

```
// Append_Liste A_List_Type als ADT ========
#ifndef __INC_A_LIST_TYPE__
#define __INC_A_LIST_TYPE__

class A_List_Type {   // Interface
// append verlaengert die Liste hinten;
// reset setzt einen Lesezeiger auf das
// vordere Ende;
// read liest ein Element und setzt den
// Lesezeiger auf das naechste Element;

public:
   A_List_Type();       // Konstruktor
   void append(char *x);// x wird kopiert
   void reset();        // zuruecksetzen
   void read(char *&x); // lesen, weiters.
   bool nonempty();     // fuer read anfangs
   bool nonfull();      // fuer append
   bool is_end();       // fuer read

private:
   unsigned int const N=999;
                        //Behaeltergroesse
   int Last;   // Index letztes Element
   int Read;   // Index aktuelles Element
   char *bc[N];// Behaelter als Feld
   bool nempty;
   bool nfull;
};

#endif
// end A_List_Type Interface ---------------
```

Sequentielle Realisierung von A_List_Type

– Realisierungsskizze:




```
// A_List_Type body ----------------------
A_List_Type::A_List_Type() {
   Last = 0; Read = 0;
   nempty = false; nfull = true;
}

void A_List_Type::append(char *x) {
   if (Last < N) {
      bc[Last] = x; Last++;
      nfull = (Last<N); nempty = true;
   }
}
...

void A_List_Type::read(char *&x) {
   if (Read != Last) {
      x = bc[Read]; Read++;
   }
}
...
// end A_List_Type body ====================
```

Diskussion

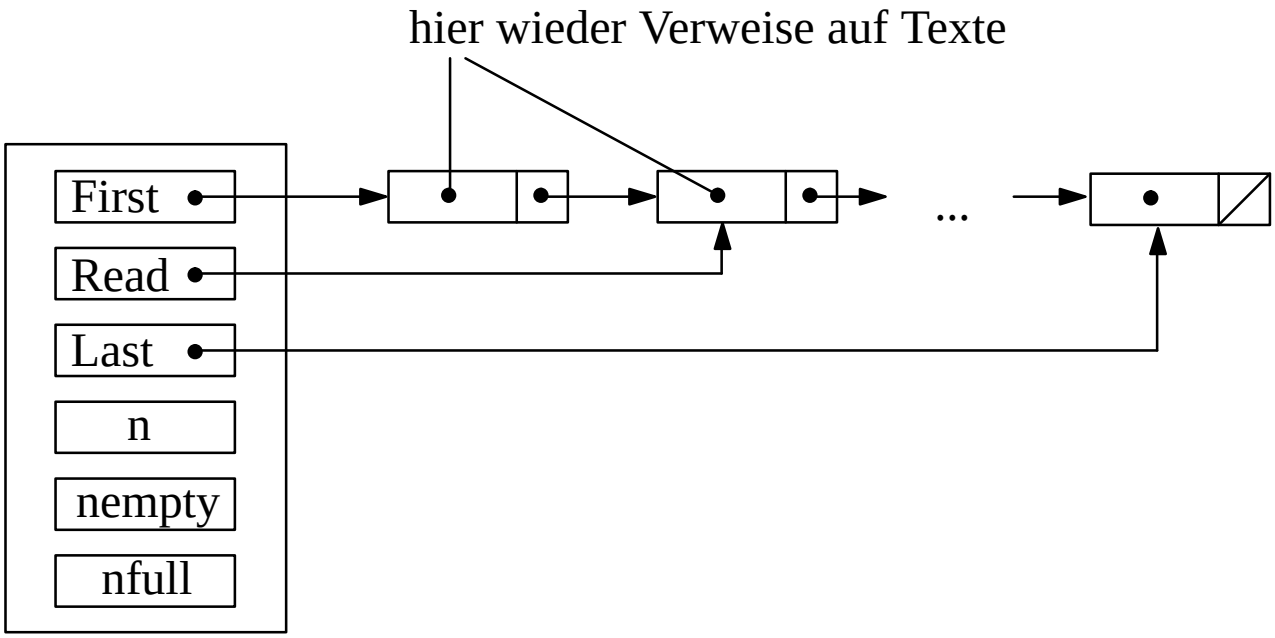
– `append`, `read`: O(1)

– sequentielle Realisierung geeignet
  wenn nur `append(el)` benötigt wird
  wenn die Gesamtzahl der Elemente zur Entwicklungszeit bekannt

– weitere mögliche Operationen von `A_List_Type`
  (später nicht benötigt)
  `read(p)`: lies Element an Position `p`:
    sequ. Realisierung mit Feld erweiterbar: O(1)
  `insert(el, p)`: Füge `el` an Position `p` ein:
    Verschieben des entspr. Teilfelds nach unten: O(n)
  `delete(p)`: Löschen des Elements an Position `p`:
    Verscheiben des entspr. Teilfeldes nach oben: O(n)
    insb. `delete(1)`: Verschieben aller Elemente
    `deletefirst`
  (vergleiche jedoch zirkuläre Speicherung von oben)

– können `insert(el,p)`, `delete(p)` bzw.
  `deletefirst()` verbessert werden?

– Realisierungsskizze



– Änderung des privaten Teils der Schnittstelle
Hinzunahme von `deletefirst()`

```
public:
   B_List_Type(); // public-Teil genau
   ...            // wie A_List_Type
   void deletefirst();
   ...
private:
  struct B_List_item { // Struktur der
    B_List_item *next; // Listenelemente
    char        *item;
  };
  B_List_item *First;  // Struktur des
  B_List_item *Last;   // Verwaltungsblocks
  B_List_item *Read;
  bool nempty;
  bool nfull;
};
```

– Änderung des Rumpfes der Liste

```
#include ”blist.h”

B_List_Type::B_List_Type() {
   First = NULL; Last = NULL; Read = NULL;
   nempty = false; nfull = true;
}

void B_List_Type::deletefirst() {
   if (First != NULL) {
      B_List_item hilf = First;
      First = First–>next;
      delete hilf;
   }
}

void B_List_Type::append(char *x) {
   // Speicher holen fuer neues Element
   B_List_item *newitem = new B_List_item;

   // neues Element initialisieren
   newitem–>next = NULL;
   newitem–>item = x;

   // und an die Liste anhaengen
   if (! nempty) { // Liste noch leer
      First = newitem;
      nempty = true;
   } else {
      Last–>next = newitem;
   }
   Last = newitem;
};
```

```
void B_List_Type::reset() {
   Read = First;
}


void B_List_Type::read(char *&x) {
   if (Read != NULL) {
      x = Read->item;
      Read = Read->next;
   }
}


bool B_List_Type::nonempty() {
   return nempty;
}


bool B_List_Type::nonfull() {
   return nfull;
}

bool B_List_Type::is_end() {
   return (Read == NULL);
}


// end body B_List ----------------------
```

Diskussion Zeigerrealisierung

– Aufwand `deletefirst`: O(1)
  Preis: Speicherplatzaufwand für Zeigerfelder
  Tradeoff: Laufzeit–Speicherplatz

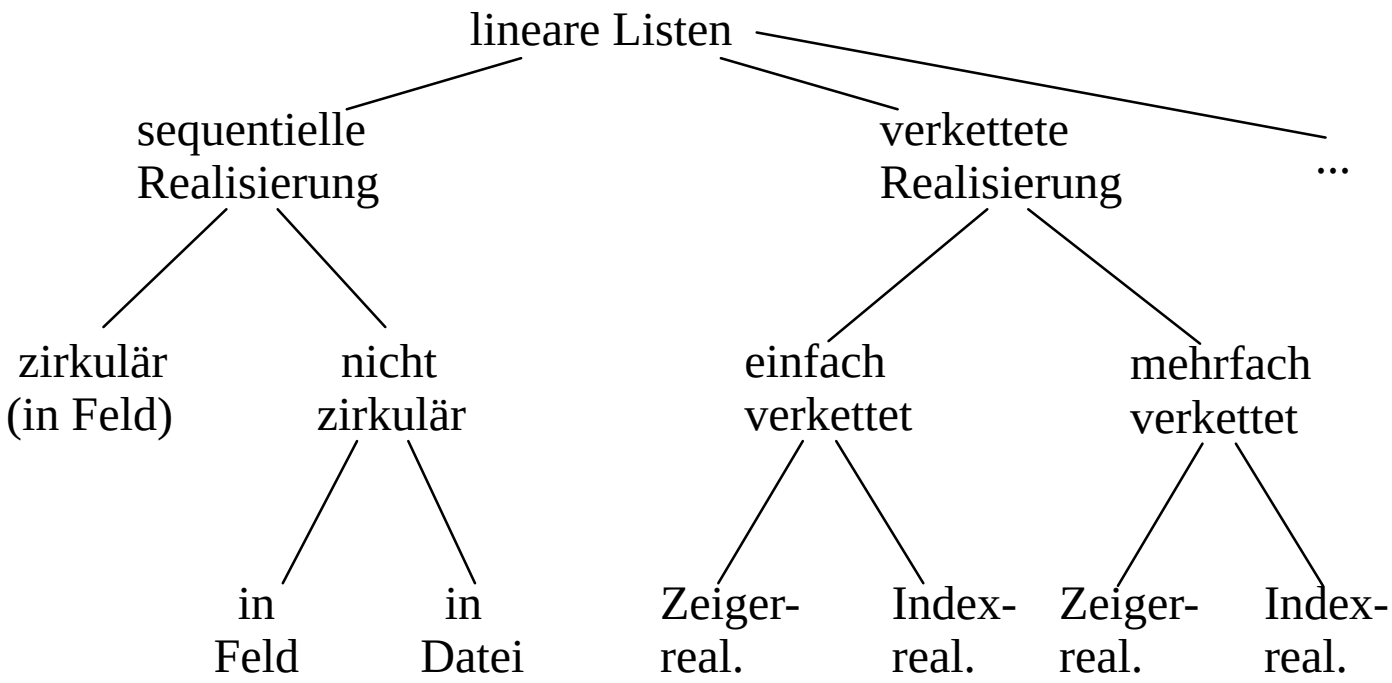
– Aufwand `delete(p)`: O(n)
  wegen Bewegung an die richtige Stelle

– wurde mit `read` bereits diese Navigation gemacht,
  dann Aufwand O(1)

– Wahl der Realisierung hängt von gewünschten Zugriffsoperationen ab:
  Generalthema dieses Teils der Vorlesung

– Bei Löschung eines Elements muß man vor dem Element stehen:
  um diese Besonderheit zu vermeiden und nun auch zurücklaufen zu können (z.B. `read_prev`) geht man besser zu doppelt verketteten Listen über (Übungsaufgabe)

Übersicht: Realisierungstechniken für lineare Listen



weitere Realisierungsarten für Listen geordnet nach Schlüsseln:

- Assoziativspeicher
- Binärbaum
- etc.

} z.T. später

Verwendung linearer Listen

Benötigte Listenarten

– Brauchen lineare Listen mit Anfügen hinten und Lesen von vorne (haben hierfür sequ. und verkettete Realisierung kennengelernt)

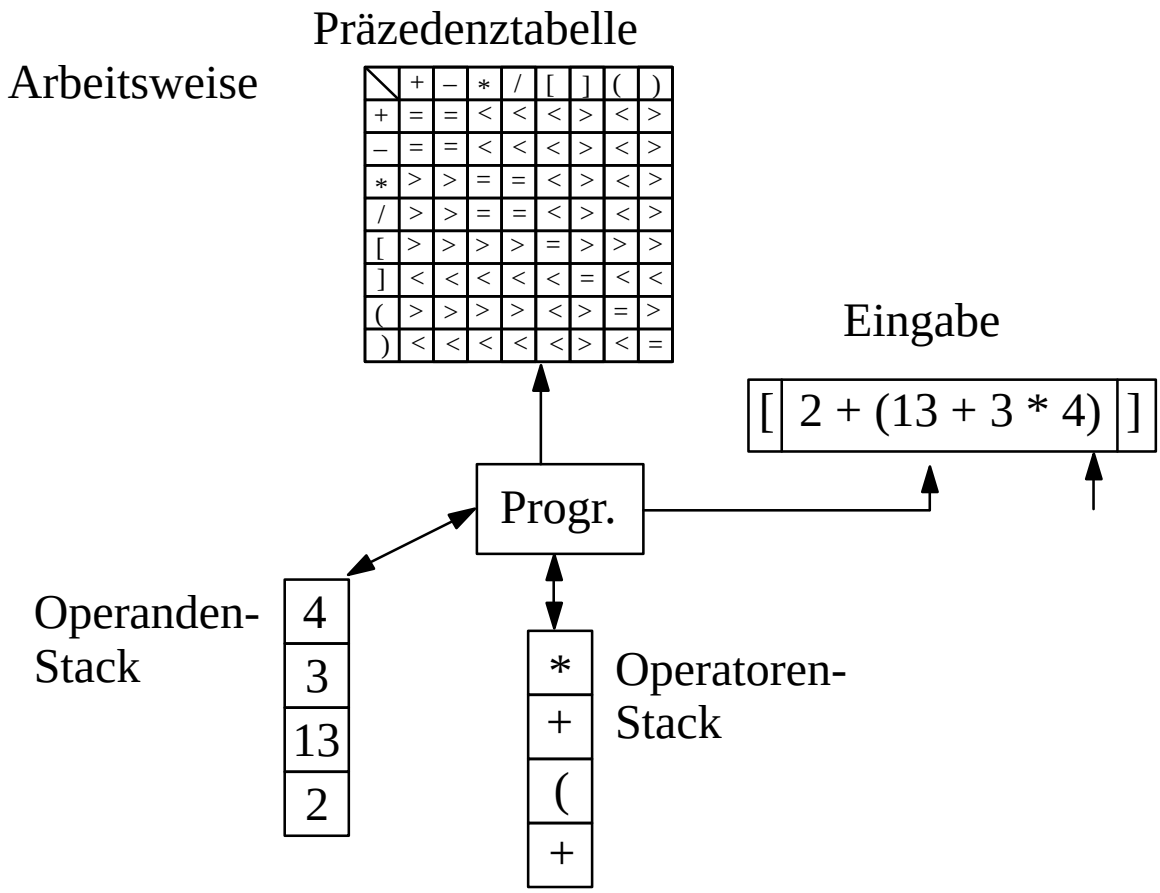
– Brauchen lin. Listen als LiFo-Datenstruktur (Keller, Stapel, Stack) mit Texten als Element

```
// ADT Stack_Type Interface ================
#ifndef __INC_STACK_H__
#define __INC_STACK_H__

class Stack_Type {
// ...
public:
   Stack_Type();
   void push(int x);
   void pop();
   void read_top(int &x);
   bool isempty();
   bool isfull();

private:

};
#endif
// end Stack_Type Interface ----------------

// body Stack_Type ------------------------
// wieder sequ. oder verkettet(Uebung)
...
// end body Stack_Type ====================
```

Aufgabe: Auswertung einfacher ganzzahliger Ausdrücke

– Ausdrücke in Infixnotation enthalten:

Literale:	13, 40
binäre Operatoren:	+, –, *, /
Begrenzer:	(,)



\	+	–	*	/	[	]	(	)
+	=	=	<	<	<	>	<	>
–	=	=	<	<	<	>	<	>
*	>	>	=	=	<	>	<	>
/	>	>	=	=	<	>	<	>
[	>	>	>	>	=	>	>	>
]	<	<	<	<	<	=	<	<
(	>	>	>	>	<	>	=	>
)	<	<	<	<	<	>	<	=

– Tokens sind:

PL=–1; MI=–2; MU=–3; DI=–4; BO=–5;BC=–6;
Start=–7; END=–8
oder ganzzahlige Werte für Literale

Brauchen `scan`, das Token als ganzz. Wert zurückliefert,
d.h. entweder Wert für Begrenzer/Operator
oder int-Wert für ganzzahliges Literal

Brauchen Pozedur `precendence`, die für zwei Operatoren/Begrenzer Operatorenvorrang bestimmt, also z.B.

+, *	$\lessdot$
+,+	$\doteq$
*,+	$\gtrdot$

```
#include <iostream>

#include ”alist.h”
#include ”stack.h”

enum OpDel {
  PL=–1, MI=–2, MU=–3, DI=–4, BO=–5, BC=–6,
  START=–7, END=–8
};

enum ComRes {GT, EQ, LT}; // fuer Praez.Vergl.

void Scanner(char *ausdruck, A_List *token);
int Auswerten(A_List *token);

int main(int argc, char *argv[]) {
  int wert;
  char *ausdruck = argv[1];
  A_List *token = new A_List();
  token–>append(START);
  // virtueller Operator mit hoechster
  // Prioritaet als Anfangszeichen
  Scanner( ausdruck, token );
  // Erzeugt aus dem Ausdruck eine Liste
  // von Token
  token–>append(END);
  // virtueller Operator mit niedrigster
  // Prioritaet als Endezeichen
  wert = Auswerten( token );
  cout << ”Der Wert des Ausdrucks ”;
  cout << ausdruck << ” ist ” << wert;
  cout << endl;
  delete token;
}
```

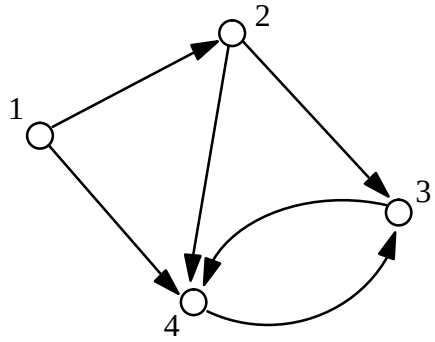
```
int Scan(char *token);
void Scanner(char *ausdruck, A_List *token){
...
};
ComRes precedence(int o1, int o2);

int Auswerten(A_List *token) {
  int wert, t;
  int lastop=START;

  Stack_Type *op = new Stack_Type();
  Stack_Type *lit = new Stack_Type();

  token->read(t);
  while (true) {
    if (t >= 0) {
    // Token t ist ein Literal
      lit->push(t);
      // und kommt auf den Literalstack
      token->read(t);
    } else { // sonst ist t ein Operator
      int curop = t;

      // Abbruchbedingung der Endlosschleife
      if (START==lastop && END==curop) {
        break;
      }

      if (LT==precedence(lastop, curop)) {
      // aktueller Operator hat groessere
      // Praezedenz und kommt auf den Stack
        op->push(curop); lastop = curop;
        token->read(t);
```

```
        } else {
        // der letzte Operator auf dem Stack
        // kann ausgewertet werden
          int l1, l2;

          switch( lastop ) {
          case START:
            op->push(curop); lastop=curop;
            token->read(t); break;
          case BO: // '('
            if (BC==curop) {
            // Klammern weglassen
              op->pop(); op->read_top(lastop);
              token->read(t);
            } else {
              op->push(curop); lastop=curop;
              token->read(t);
            }
            break;
          case PL: // '+'
            op->pop(); op->read_top(lastop);
            lit->read_top(l2); lit->pop();
            lit->read_top(l1); lit->pop();
           lit->push(l1+l2); break;
          // MI, MU, DI entsprechend
          ...
          }
        }
      }
    }
    // auf dem Literalstack liegt das Ergebnis
    lit->read_top(wert);
    delete op;
    delete lit;
    return wert;
}
```

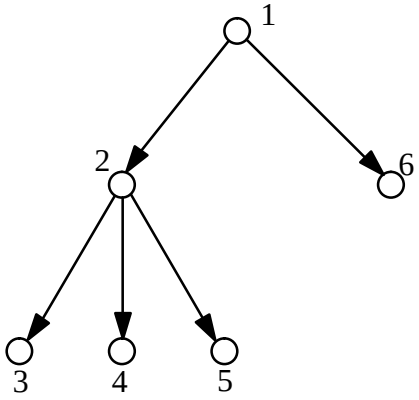
Bäume

Allgemeines

– Def.
gerichteter Graph G ist G = (K, E) mit
K endl. Menge (*Knoten*)
E $\subseteq$ K x K (*Kanten*)
k_2 *Nachf.* von k_1
k_1 *Vorg.* von k_2
Folge von Knoten
Kz = (k_1, ..., k_{n+1}) heißt (gerichteter) *Kantenzug*
g.d.w. (k_i, k_{i+1})$\in$E
n *Länge* von Kz, Anzahl der Kanten
Zyklus g.d.w. Kantenzug geschlossen, d.h. k_1=k_{n+1}
(Auslauf) *Grad* von k ist Anzahl der Nachfolgerknoten
(weiteres später)



– Def. (*Wurzel*) *Baum* ist zyklenfreier Graph, wobei jeder Knoten höchstens einen Vorgänger hat und bei dem jeder Knoten von bestimmten Knoten (Wurzel) über Kantenzug erreichbar ist.
(Def. *Vater, Söhne, Brüder, Teilbäume, Blätter, innere Knoten, binärer Baum, ternärer Baum*)

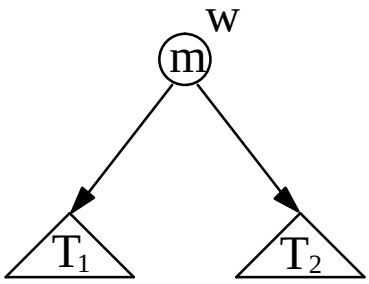


– Def. Sei k Knoten eines Baumes T.
  Stufe (Niveau) von k def. durch

$$lev(k) = \begin{cases} 1, & k \text{ ist Wurzel} \\ lev\,(Vorg\,(k)) + 1, & \text{sonst} \end{cases}$$

  Höhe $h\,(T) := \max_{k \text{ Knoten von } T} (lev\,(k))$

– Def. (Baumdurchläufe binärer Bäume):
  • Baum leer bzw. besteht der Baum nur aus isolierten Knoten, dann ist die leere Liste bzw. die Liste aus dem einz. Knoten die Preorder-, Inorder- und Postorder-Liste



  • Sonst ist
    Preorder-Durchlauf Pr(T) def. durch $Pr(T):=(m, Pr(T_1),Pr(T_2))$
    Inorder- In(T) $In(T):=(In(T_1),m,In(T_2))$
    Postorder- Po(T) $Po(T):=(Po(T_1),Po(T_2),m))$

    mit $m = f_{lab}(W)$

– Bem.: Syntaxbaum Inorder-Durchlauf ⇒ Infixnotation
  Preoreder- ⇒ Präfixnotation
  Postorder- ⇒ Postfixnotation

ADT Baum und Verwendung

ADT Baum: Schnittstelle

– Operationen eines ADT Tree
(bel. geordn. mark. Baum):

`Parent(n,T)` liefert Vater von n in T, für Wurzel undef.

`leftmostChild(n,T)` liefert ersten Nachfolger von n in T; undef. für Blatt

`rightSibling(n,T)` liefert nächsten Bruder; undef. für letzten Sohn

`Label(n,T)` liefert Markierung von n

$\texttt{Create}_i(m, T_1, T_2, \ldots, T_i)$ liefert Baum mit Wurzel k, die mit m markiert ist, und T_1, T_2, ... als Nachfolger von links nach rechts besitzt. Spezialfall i=0 erzeugt isolierten Knoten.

`Root(T)` liefert Wurzelknoten, undef. für leeren Baum

`MakeNull(T)` macht T zum leeren Baum

`isEmpty(T)` liefert `true` für leeren Baum, sonst `false`

– Operationen eines ADT BinTree

`LeftTree(n,T), RightTree(n,T)`

...

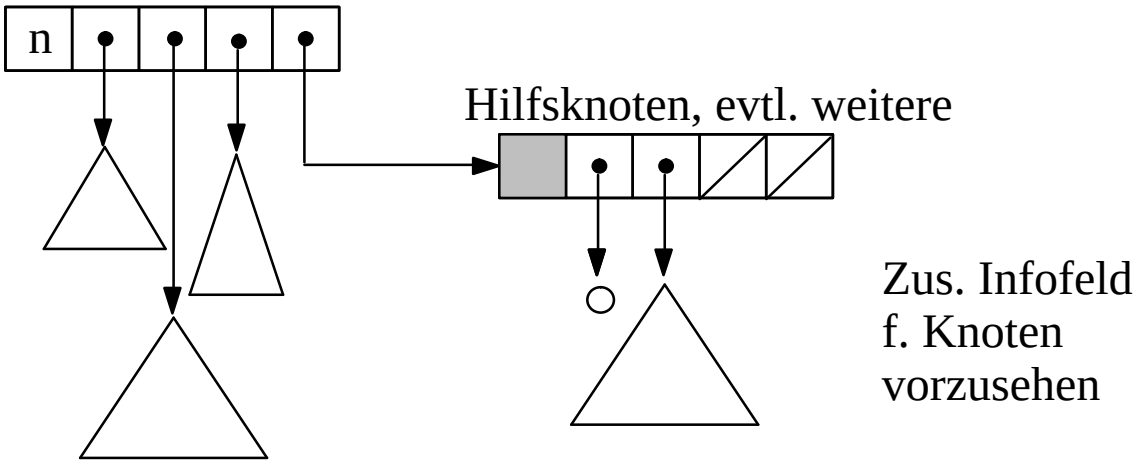
```
// nichtrekursive Fassung des
// Preorder–Durchlaufs
void NRPreorder(Node n, Tree t) {
  Node m;
  NodeStack s;
  // ADO mit Eintraegen des Typs Node,
  // enthaelt jeweils den Pfad von der
  // Wurzel zu aktuellem Knoten
  if (!isempty(t)) {
    m = n;
    while (true) {
      if (m!=NULL) {
        Ausgabe(Label(m, t));
        push(m, s);
        // gehe zum naechsten Nachfolger
        // ueber
        m = leftmostChild(m, t); }
      else {
        // Abarbeitung des Pfades auf dem
        // Keller ist jetzt fertig
        if (isEmpty(s)) { return; };

        // fahre mit Geschwisterknoten des
        // obersten Kellerelements fort:
        m = rightSibling(read_top(s), t);
        pop(s);
      }
    }
  }
}
```

– noch effizientere nichtrekursive Fassung gewinnt man, wenn man den Keller selbst im Baum ablegt.
  Hierzu muß
  - der ADT Tree auch die Op. Parent (Übergang zum Vater) zur Verfügung stellen (war oben nicht nötig),
  - Verwaltungsinfo am Knoten abgelegt werden, wie z. B. “schon besucht”

Realisierungsvarianten für den ADT

Listenelemente für Knoten und Nachfolger

– Knoten und Verweise auf Nachfolger werden als Listenelemente abgelegt.
  Hat jeder Knoten festen Auslaufgrad: entsprechende Knotenelemente, sonst Überlaufelemente



– Realisierung mit C++ - Zeigern (oder Indizes), ersteres etwa:

```
struct Knoten;
typedef Knoten* ZK;
struct Knoten {
   unsigned Anzahl;
   item Info;
   ZK Zeiger[4];
};
```

– damit die Operationen leftmostChild, rightSibling, Label, $Create_i$ leicht zu realisieren
  Operation MakeNull: Freigeben aller Knotenelemente

– Speicherplatzeffizienz:
  Sei T ein Baum vom Grade k mit n Knoten, realisiert durch Knoten mit je k Zeigerfeldern

  $\Rightarrow n*k - (n - 1) = n*(k - 1) + 1$ sind frei

  Zeigerdef. Anz. Kanten

  damit

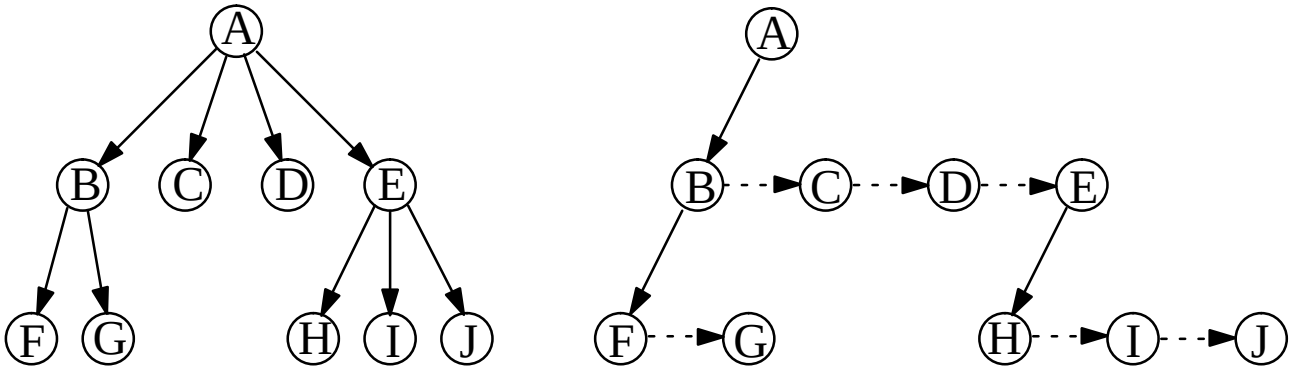
  ternäre Bäume: mehr als 2/3 der Zeiger NIL

  binäre Bäume: Hälfte NIL

– Bilanz noch einmal verbesserbar durch anderen Datentyp für Blatt (ohne Zeiger auf Nachfolger):
  Dann sind Erweiterungsoperationen etwas komplizierter.

Abspeicherung eines bel. Baumes durch einen binären

– Umwandlung interessant wegen obiger Speicherplatzüberlegung:

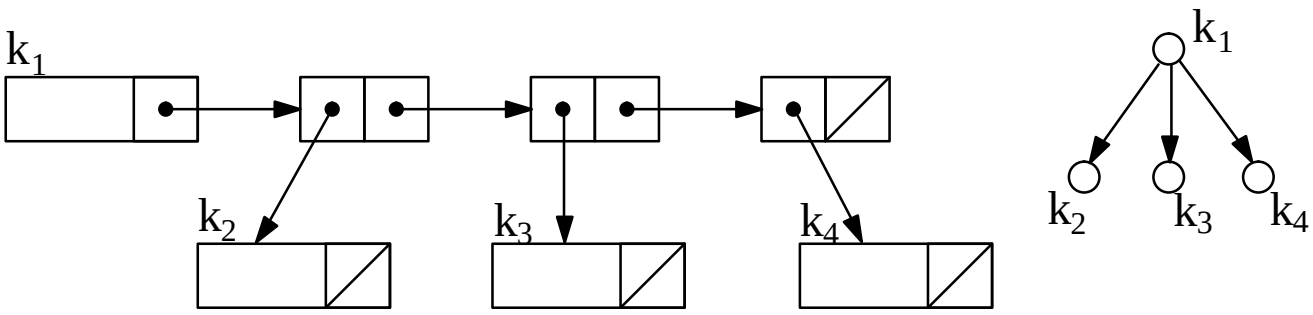


2+4+4+
1+4+1+4+4+4=31 NIL-Felder
bei 4 Zeigerfeldern

11 NIL-Felder

sog. leftmostChild-rightSibling-Implementation

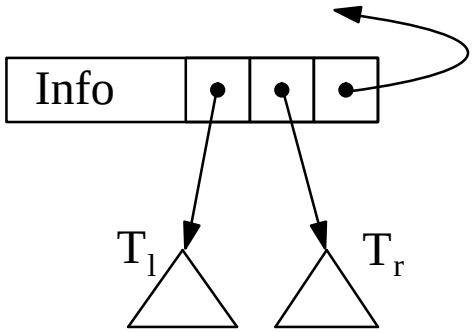
– andere Implementationstechnik: zusätzlich Kantenelemente



Speicherbilanz nicht günstiger als bei obiger Binärbaumrealisierung:

n+(n–1)*2 gegenüber 2n Zeigerfeldern

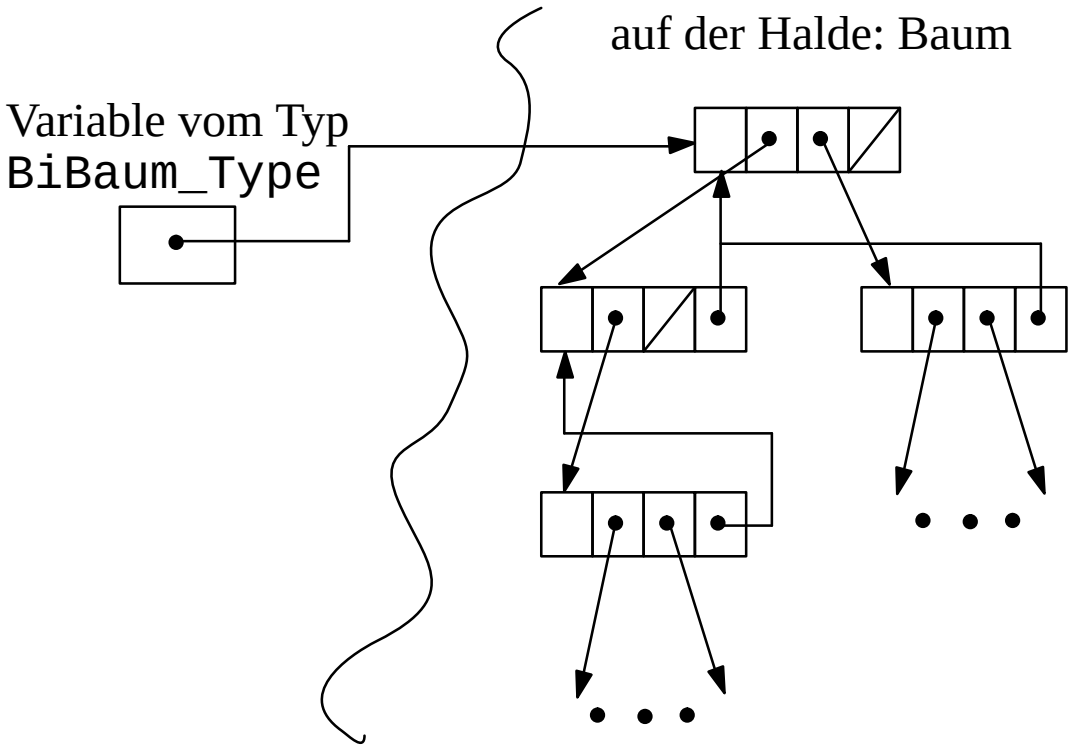
– alle bisher besprochenen Implementationen erlauben keine effiziente Implementation der Operation Parent
  Lösung: Zusätzlicher Zeiger bei jedem Knoten auf den Vater.
  Für Binärbaum mit Zeigerrealisierung etwa:



– Struktur der Knotenelemente:

```
struct Knoten;

typedef Knoten* ZK;

struct Knoten {
   item Info;
   ZK liBaum;
   ZK reBaum;
   ZK Vater;
};

typedef ZK BiBaum_Type;
```

Bei ADT als Klasse ist der Typ die Klasse selbst (Übung!)
Wie sieht die Lösung aus, wenn auch `Knoten` zu Klasse wird (Übung!)



– Resumee:

Leftmost_Child-Right_Sibling-Realisierung günstiger als
Knotenelemente mit Überlaufelementen
besser als Knoten- und Kantenelemente
ggfs. mit Verfädelung
Realisierung auf Halde oder mit Indizes (Cursorrealisierung)

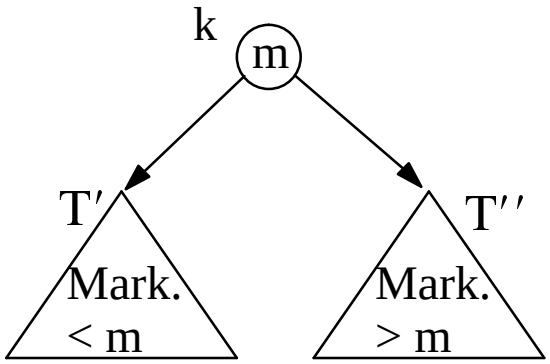
Verwendung binärer Bäume als Suchbäume

Binäre Suchbäume

– Def. binärer Suchbaum ist bin. mark. Baum T = (K,E) mit

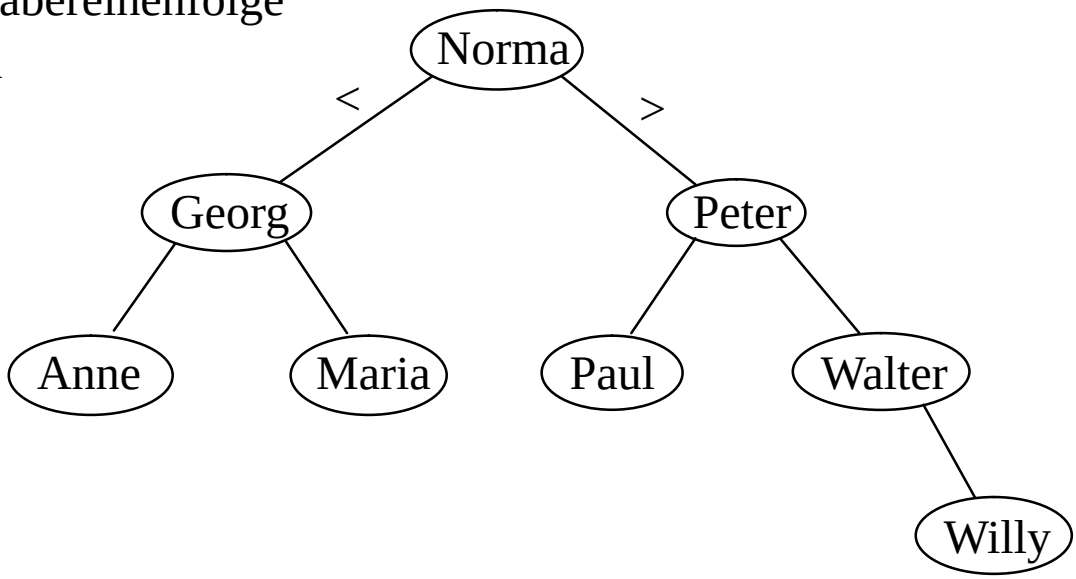
$$\forall\, k \in K \text{ gilt}$$
$$\begin{cases} f_l(k') < f_l(k) \;\; \forall k' \in K' \text{ mit } LeftTree(k,T) = T' = (K',E') \\ f_l(k'') > f_l(k) \;\; \forall k'' \in K'' \text{ mit } RightTree(k,T) = T'' = (K'',E'') \end{cases}$$

$f_l(k)$ liefere die Info von k zum Vergleich (label)



– Bsp. Eingabereihenfolge

Norma
Peter
Paul
Georg
Walter
Maria
Anne
Willy



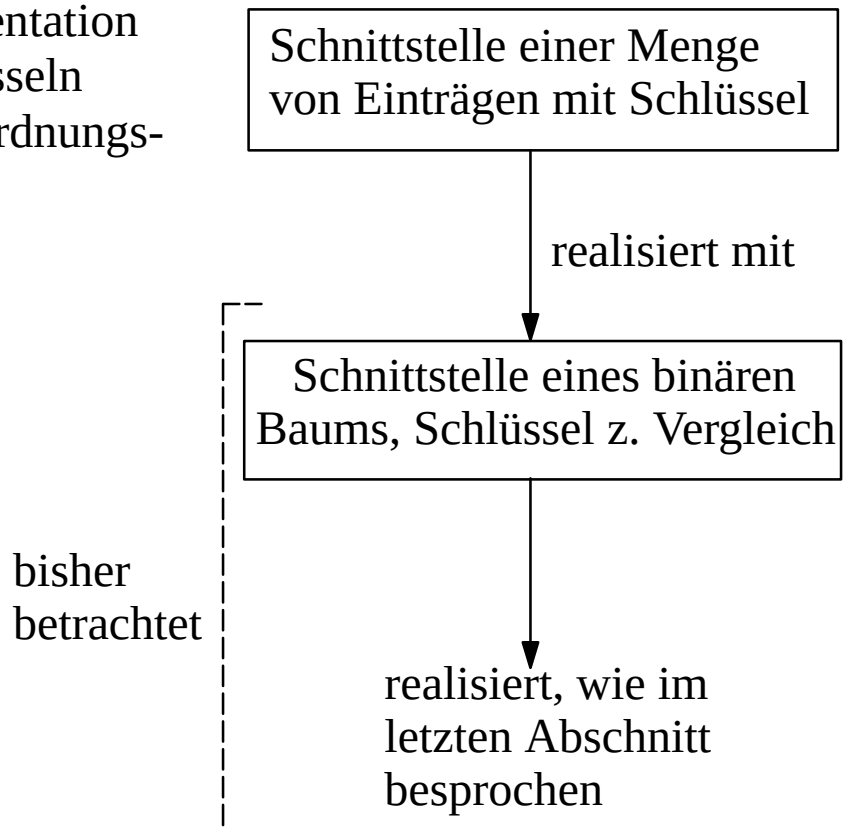
– Bem.

- < bez. sich auf ganzz. Schl., oder lexikograph. Ordnung auf Zeichenketten
- Inorder-Reihenfolge liefert totale Ordnung
- Baumaufbau (”Ausgeglichenheit”) hängt von der Eingabereihenfolge ab: z.B. liefert Eingabe in totaler Ordnung degenerierten Baum (Pfad)

Menge realisiert mit binärem Suchbaum

– jetzt “inkrementelles” Sortieren: Bei jeder Veränderung wird gleich der veränderte Teil neu sortiert,
Suche macht von der jeweils aktuellen Sortierung Gebrauch.
Anwendung: z.B. Symboltabelle

– jetzt Baum als Implementation von Mengen mit Schlüsseln auf denen eine totale Ordnungsrelation gilt



– Realisieren
alle folg. Algorithmen mithilfe eines ADTs `BinTree_Type`, der z.B. Op. `LabelNode()`, `Root()`, `MakeNull()`, `IsEmpty()`, `rightTree()`, `leftTree()`, `InsertRightNode(m)` (als Spezialfall von $\texttt{Create}_{\texttt{i}}\texttt{(...)}$ von oben) und einen Klassenbezeichner `BinTree_Type` exportiert

Vorteil: Sind bei Alg. zur Suchbaumbehandlung vollst. unabhängig von der Realisierungsart des Binärbaums (Zeigerreal., Indexreal., etc.)

– Suche in Menge `M`, die intern als binärer Suchbaum org. ist. `Contains` ist Schnittstellenoperation eines entspr. Moduls (Klasse)

```
class A_Menge_Type { // interface
// ...

public:
   A_Menge_Type(); // Konstruktor
   bool Contains(elType x);
   ...
private:
   BiBaum_Type StTree; // Zeiger auf intern
                       // benutzten bin. Suchbaum
};
...

bool A_Menge_Type::Contains(elType x) {
   BiBaum_Type B=StTree;
   while (B!=NULL) {
      if (B->LabelNode() > x) {
         B=B->rightTree(); }
      else if (B->labelNode() < x) {
         B=B->leftTree(); }
      else {
         return true; // gefunden
      }
   }
   return false; // nicht gefunden; stehen
                 // auf Knoten unterhalb dem
                 // einzufuegen waere
}
```

– Analyse Suchen
  - für ausgegl. Suchbaum $\in$ $O(\log_2 n)$
    (n Anz. d. Einträge = Anz. d. Knoten des Suchbaums)

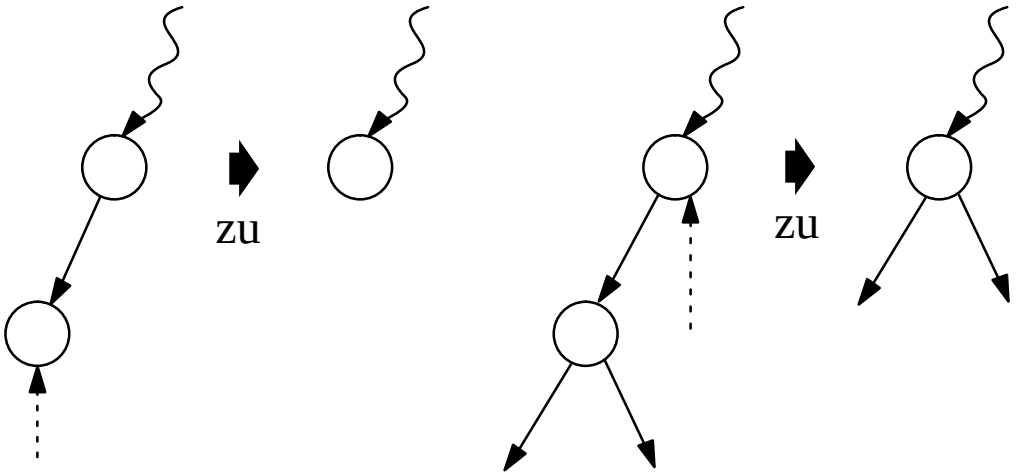
  - für degenerierten Suchbaum $\in O(n)$

  - beweisbar: deg. Baum selten. Bei zufäll. Reihenfolge:
    $$C_{\emptyset} = 2 \ln n$$
    (Was heißt zufällig bei alphanumerischen Schlüsseln?)

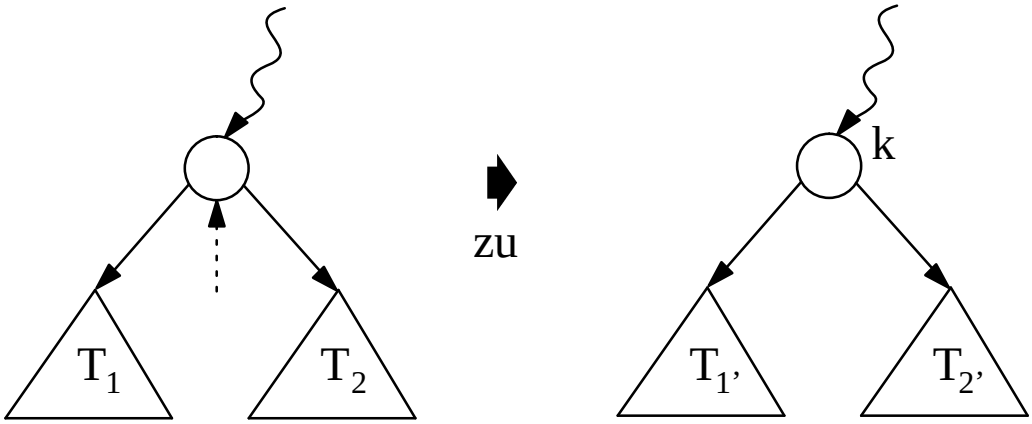
  - Komplexitätsaussage gilt auch f. Einfügen, Löschen und Minimumssuche

– im Rumpf von Menge als lok. Prozedur
  Binärbaum-Schnittstelle muß entspr. Hilfsmittel anbieten

– Löschen in binärem Suchbaum
  • Vorüberlegungen:
    a) einfach, falls Blatt oder innerer Knoten mit nur einem Nachfolger



    b) Knoten mit zwei Nachfolgern

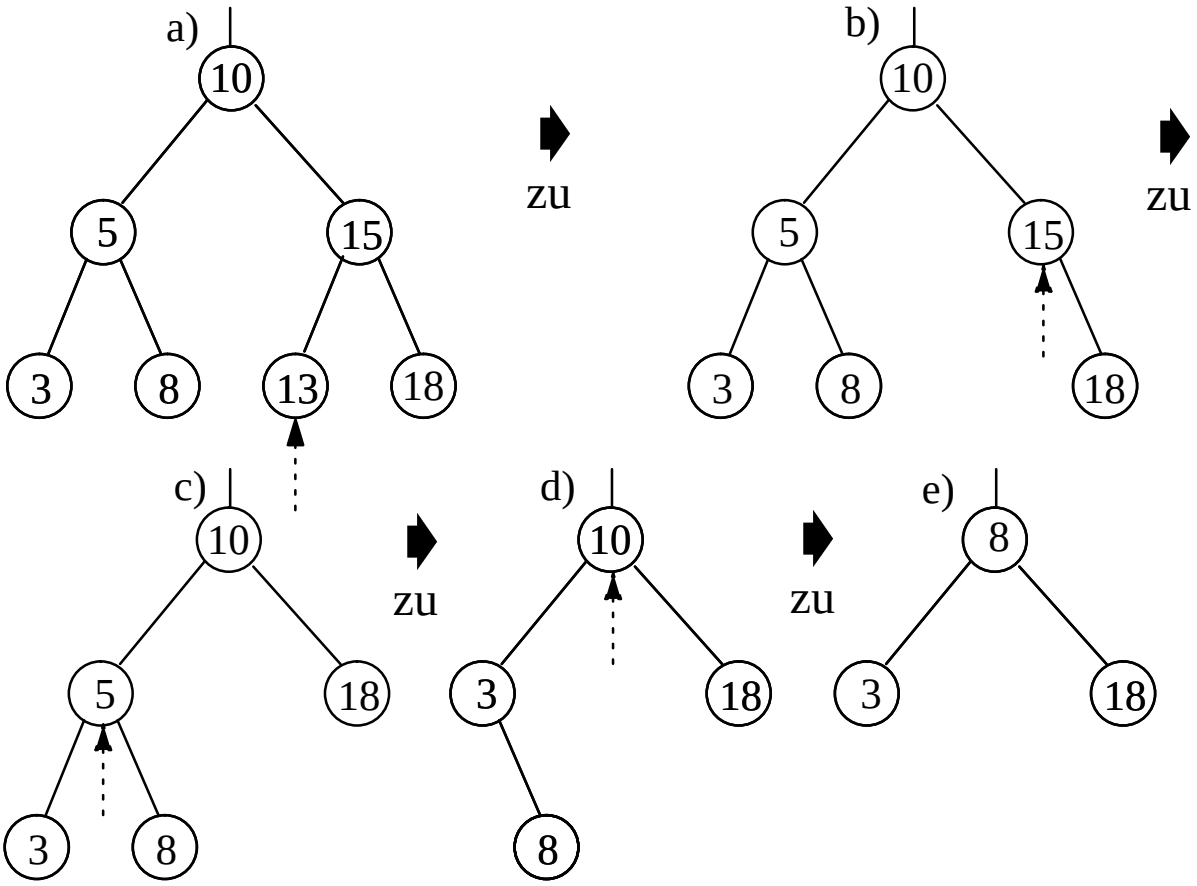


2 Mögl.: a) k ist gößtes El. aus T_1
dann ist $T_2' = T_2$

b) k ist kleinstes El. aus T_2
dann ist $T_1' = T_1$

kleinstes El. aus T_2 oder größtes aus T_1 haben höchstens einen Nachfolger.

– Beispiele

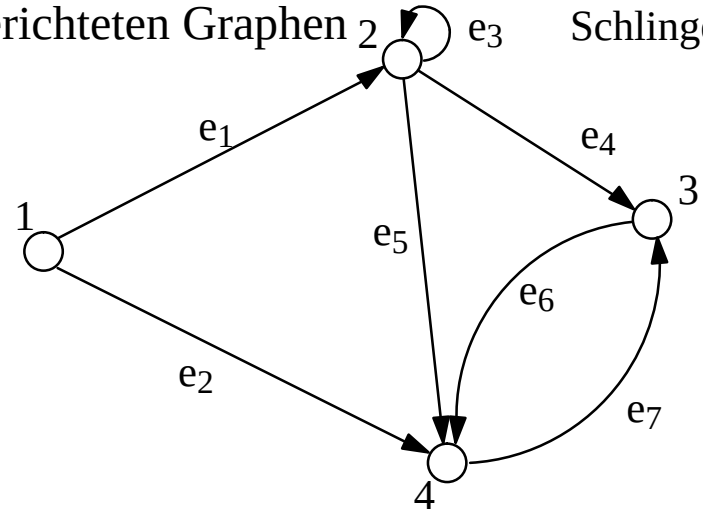


– Einfügen problemlos
  unter bisheriges Blatt
  unter Knoten mit bisher nur anderem Nachfolger
  erzeugt ggfs. Zwischenknoten mit nur einem Nachf.

Graphen und Graphprobleme

Definitionen und Beispiele

– Beispiel eines gerichteten Graphen



K={ 1, 2, 3, 4 }

E={ (1, 2), (1, 4), (2, 2), (2, 3), (2, 4), (3, 4), (4, 3) }
e_1 e_2 e_3 e_4 e_5 e_6 e_7

– Def.

k1, k_2 *benachbart* $k_1 \vdash\dashv k_2$ g.d.w. $(k_1, k_2) \in E$ oder $(k_2, k_1) \in E$

Nachbarknoten NeighN(k) := {k' | k' ∈ K ∧ k ⊢⊣ k'}

$NeighN^-(k)$ Nachbarn über einl. Kanten

$NeighN^+(k)$ Nachbarn über ausl. Kanten

inzidierende Kanten IncE(k):={ e | e ∈ E ∧ ∃ k' ∈ K
(e= (k, k') ∨ (k', k)) }

$IncE^-(k)$ einl. Kanten

$IncE^+(k)$ ausl. Kanten

(Kanten)Grad von k (abgek. deg(k)) ist | IncE(k) |

– Beispiel

In G gilt

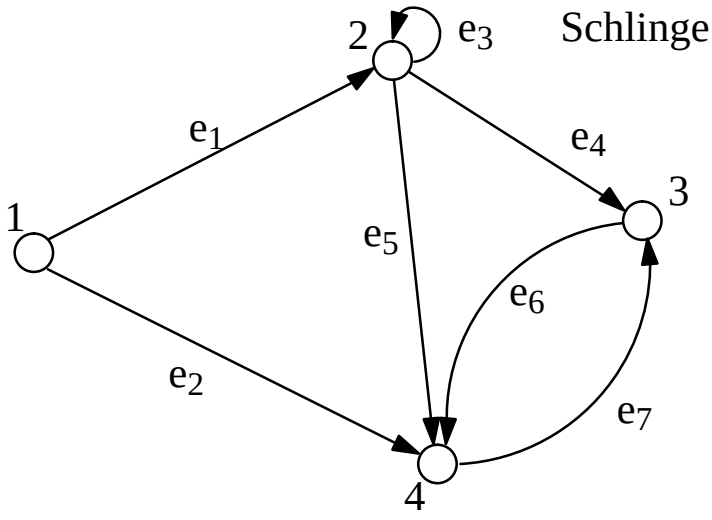
$2 \vdash\dashv 3$

$NeighN(2) = \{1, 2, 3, 4\}$

$NeighN^{-}(4) = \{1, 2, 3\}$

$IncE^{-}(3) = \{e_4, e_7\}$

$deg(2) = 4$



G

– Def. *markierter Graph* $G = (K, E, f_{nl}, f_{el})$ mit
$G' = (K, E)$ ist Graph
$f_{nl} : K \mapsto NL$ *Knotenmark.*, f_{nl} *Knotenmarkierungsfunktion*
$f_{el} : K \mapsto EL$ *Kantenmark.*, f_{el} *Kantenenmarkierungsfunktion*

– Probleme, die auf Graphenprobleme zurückgeführt werden können:
  (Knoten: Kreuzungen, Endpunkte von Straßen, Plätze
  Kanten: Straßen
  Stadtplan, Landkarte → markierter Graph)

  Erreichbarkeitsproblem: Ist Stelle 2 von Stelle 1 aus erreichbar?
  Kürzester Weg: Welcher ist der beste Weg?
  Mehrf. Zusammenhang: Gibt es eine Verbindung, wenn eine Straße/ ein Platz blockiert ist?
  Fluß: Was ist die Gesamtkapazität des Straßennetzes zwischen zwei Punkten
  Minimalgerüst: Errichtung einer Stromversorgung z. B. für Bauernhöfe

– Bem.: Graphen zur Codierung von Sachverhalten
  Markierungen können sein:
  Knoten: Klasseneint. f. Objekte: NL endl. nichtl. Menge
  Kanten: Klasseneint. f. Bez.: EL endl. nichtl. Menge
  Knoten, Kanten: Gewichtungswerte, Kapazitäten

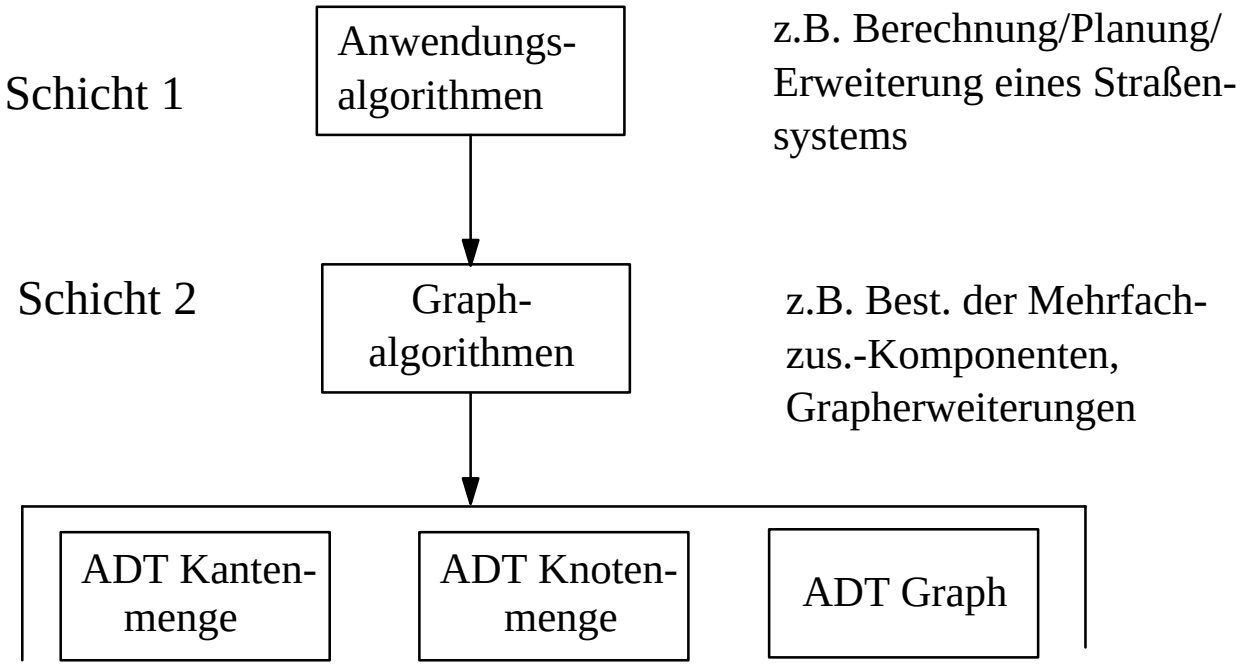
  evtl. beides:
  Knoten: Menschen,
  Klasseneint.: Angestellter, ltd. Angestellter etc
  Gewichtung z.B. Alter, Vermögen etc.

– Unterscheidung: Bezeichnung – Markierung

– ADT Graph hätte z.B. folgende Operationen

```
Label(n)
MakeNull()
isEmpty()
OutN(n, elab, NSet)
InN(n, elab, NSet)
OutE(n, ESet)
InE(n, ESet)
.
.
.
```

– daurauf aufsetzend könnten oben skizzierte Probleme programmiert werden



Vorteil: Wären in Schicht 1 und insb. Schicht 2 unabh. von spez. Real. der Graphen und der elem. Graphveränderungen

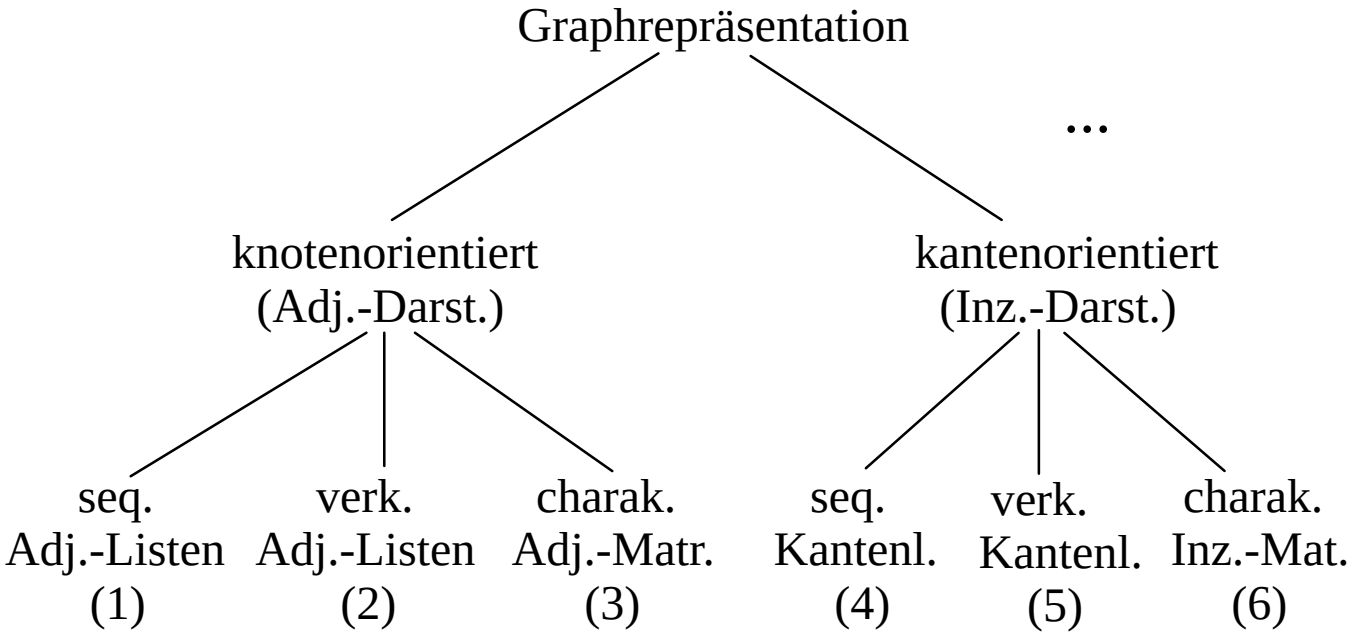
ADT Graph Realisierungsübersicht

Graphrealisierungen

– Repräsentation (Realisierungen) von Graphen
  - Zurückführung auf Mengen
    knotenorientiert (adjazenzorientiert):
      Für jeden Knoten: NeighN(k)
    kantenorientiert (inzidenzorientiert):
      Für jede Kante: Anfangsknoten, Endknoten
    ...

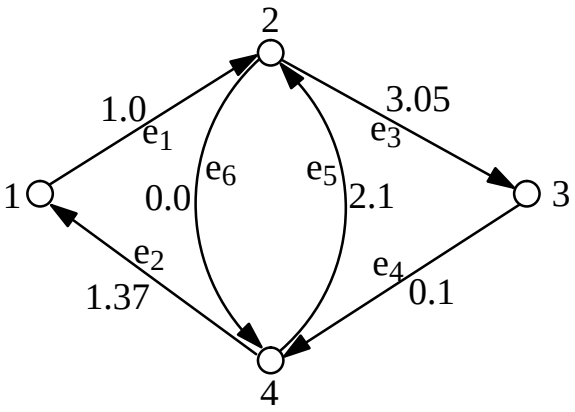
  - mögliche Realisierungen der Mengen
    sequentielle Speicherung
    verkettete Speicherung
    (Suchbaumrealisierung)
    charakteristische Speicherung

– somit mögliche Realisierungen



– Auswahl der Speicherung hängt von Operationen ab:
  z.B. sequentielle Kantenliste (Mögl. (4))
  geeignet für Graphen mit beschränkter Kantenzahl, ohne isol. Knoten

1	2	3	4	5	6	
1	4	2	3	4	2	Anfangsknoten
2	1	3	4	2	4	Endknoten
1.0	1.37	3.05	0.1	2.1	0.0	Wert d. Kante



– Einige typ. Graphoperationen (Pseudocode)
  Wollen für alle Darstellungen folg. Operationen betrachten

  - for all e in $IncE^+(k)$ do
    bearbeite alle aus k hinausg. Kanten

  - for all e in E do
    bearbeite alle Kanten

  - isEdge(k1, k2)

  dabei sei stets n Anzahl der Knoten
  m Anzahl der Kanten

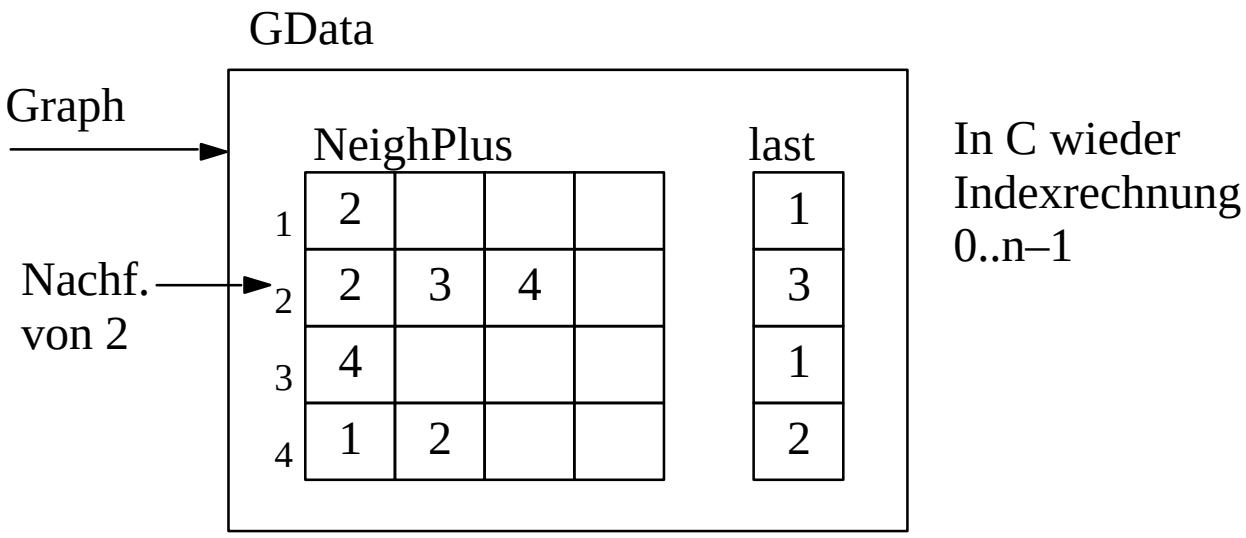
  Knotenbezeichnungen ganzzahlig

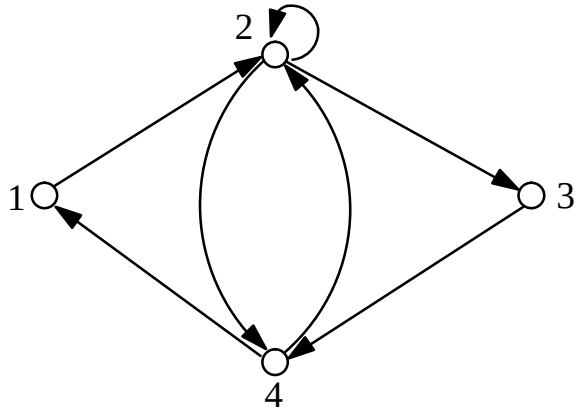
Knotenorientierte Realisierungen

– Bem.
  Zunächst ohne Markierungen
  Graph völlig durch Adjazenzangaben bestimmt
  (genügt bereits $NeighN^+(k)$ für alle $k \in K$)

Seq. Adjazenzlisten

– Bsp.:





- Realisierung

```
struct GData {
   int NeighPlus[n][n];
   int last[n];
};
typedef GData* Graph;
```

$O(n^2)$

for all e in $IncE^+(k)$ do

```
   for (int j=0; j<=last[k]; j++) {
      nimm e als (k, NeighPlus[k][j]);
      bearbeite e;
   };
```

$O(|IncE+(k)|)$

for all e in E do

```
   for(int i=0; i<n; i++) {
      for(int j=0; j<=last[i]; j++) {
         nimm e als (i, NeighPlus[i][j]);
         bearbeite e;
      };
   };
```

$O(m)$

isEdge(k_1, k_2)

```
   j=0;
   while (j<=last[k1]) {
      if (NeighPlus[k1][j]==k2) {
         return true;
      };
      j++;
   };
   return false;
```

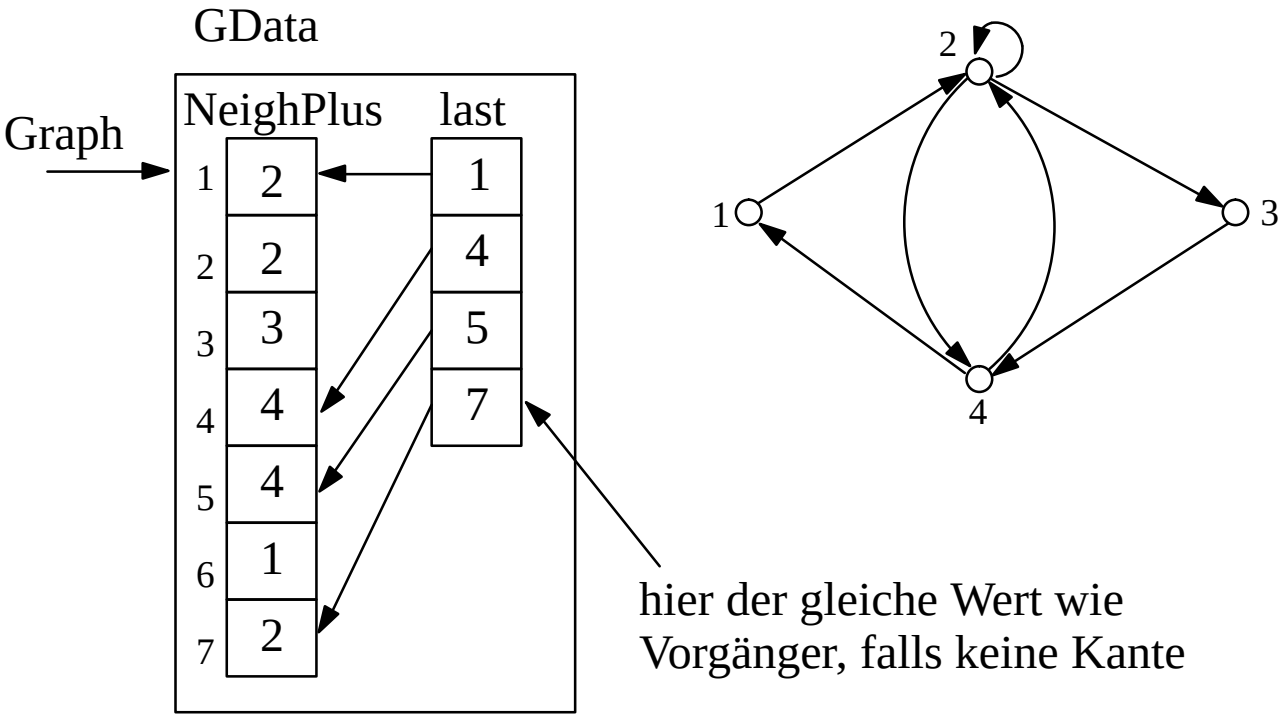
$O(|IncE+(k)|)$

- Wertung
  Hinzunahme von Kanten leicht
  Platzverschwendung enorm: der vollst. Graph tritt selten auf.
  Hinzunahme neuer Knoten schwer

Verdichtete seq. Speicherung

– Bem.:
  Falls $m << n^2$ oben viel Platz verschenkt.
  Hintereinanderlegen der seq. Knotenliste (Zeilen der Matrix)

– Bsp.



Verkettete Adjazenzlisten

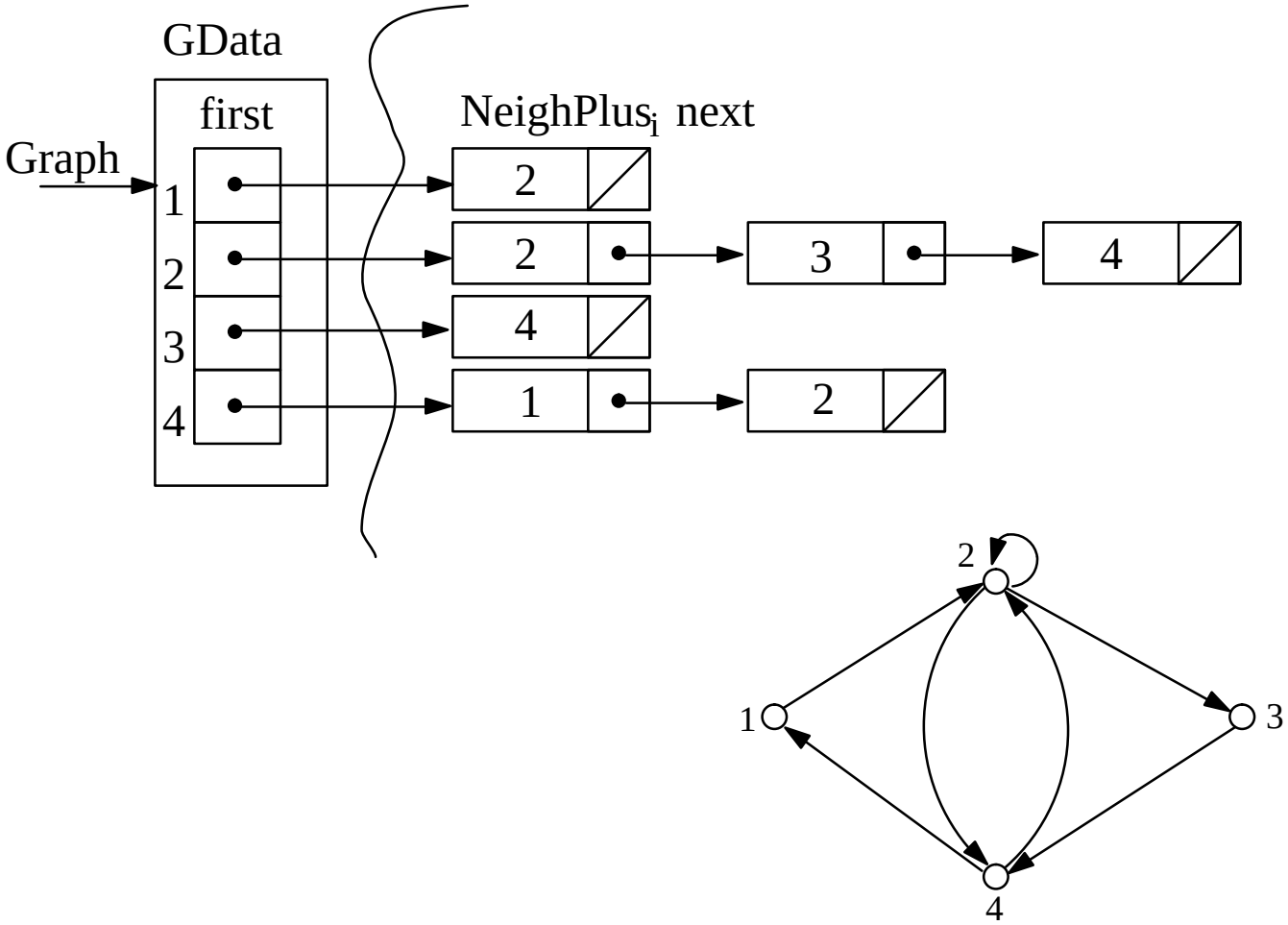
– Bem.

obg. Realisierungen durch seq. Adjazenzlisten
(verd. oder unverd.),
nicht geeignet für Graphen, die sich verändern:
Knoten einfügen/löschen
Kanten einfügen/löschen

Verkettung löst dieses Problem
Realisierung durch C++ - Zeiger und Haldenobjekte
Realisierung durch Indexverkettung

– Betrachten verzeigerte Realisierung zunächst ohne Knoten- und Kantenmarkierungen

– Bsp.



first[i] zeigt auf den Anfang der Nachfolgerliste zu Knoten i.

Lösung noch inflexibel: Knoten einfügen und löschen
⇒ Ersetzen first-Feld ebenfalls durch Haldenstruktur

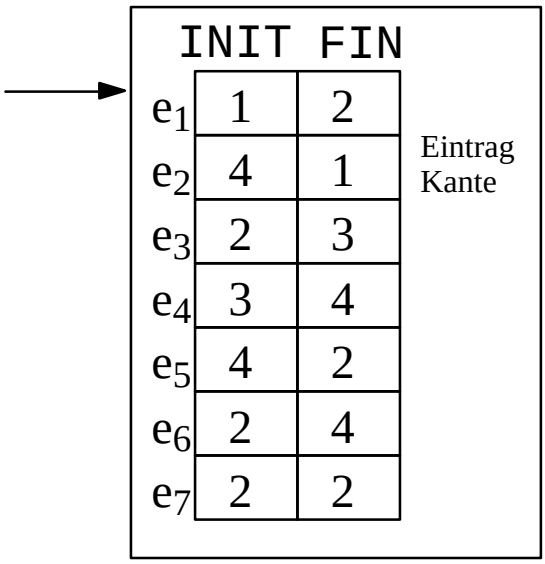
Kantenorientierte Realisierung

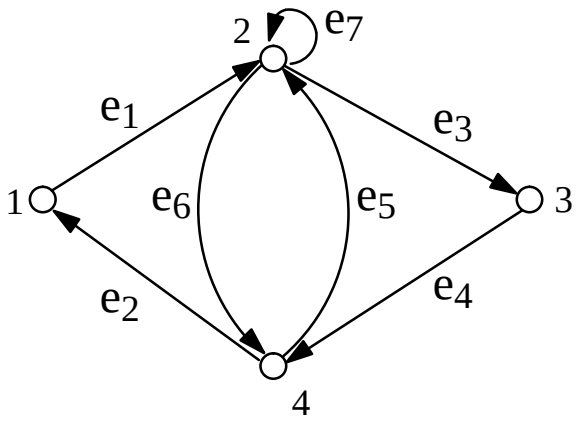
– Bemerkungen
synonym inzidenzorientierte Realisierung,
falls isolierte Knoten auftreten, zus. Abspeicherung der Knoten.

Sequentielle Kantenliste

– Beispiel

GData





Darstellung: Quellknoten, Zielknoten zu Kante,
Kantenreihenfolge bedeutungslos

- Realisierung

```
struct KantenEl {
   int Quelle;
   int Ziel;
};
typedef KantenEl GData[m];
typedef GData* Graph;
```

O(m)

for all e in $IncE^+(k)$ do

```
  for (int i=0; i<m; i++) {
     if (*G[i].Quelle==k) {
        e = (k, *G[i].Ziel);
        bearbeite e;
     };
  };
```

O(m)

for all e in E do

```
    analog
```

O(m)

isEdge (k1, k2)

```
  i=0;
  while (i<m) {
     if ((*G[i].Quelle==k1)
          &&(*G[i].Ziel==k2)){
        return true;
     };
     i++;
  };
  return false;
```

O(m)