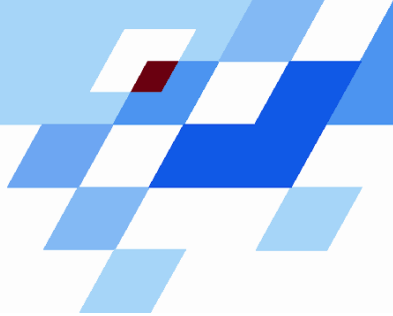


Wintersemester 2006/07

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias **Einführung in die Programmierung)**

(Vorlesung)

Prof. Dr. Günter Rudolph

Fachbereich Informatik

Lehrstuhl für Algorithm Engineering

Inhalt

- Definition ☑
- ADT Keller ☑
- ADT Schlange ☑
  - Exkurs: Dynamischer Speicher (new / delete) ☑
- ADT Liste ☑
- ADT Binärbaum ☑
  - Exkurs: Einfache Dateibehandlung ☑
  - Exkurs: Strings (C++) ☑
  - Anwendung
- ADT Graphen

} heute

ADT Binäre Bäume: Anwendung

Aufgabe:

Gegeben sei eine Textdatei.
Häufigkeiten der vorkommenden Worte feststellen.
Alphabetisch sortiert ausgeben.

Strategische Überlegungen:

Lesen aus Textdatei → `ifstream` benutzen!
Sortierte Ausgabe → Binärbaum: schnelles Einfügen, sortiert „von selbst“
Worte vergleichen → C++ Strings verwenden!

Programmskizze:

Jeweils ein Wort aus Datei lesen und in Binärbaum eintragen.
Falls Wort schon vorhanden, dann Zähler erhöhen.
Wenn alle Wörter eingetragen, Ausgabe (Wort, Anzahl) via **Inorder**-Durchlauf.

```
struct BinBaum {
  string wort;
  int anzahl;
  BinBaum *links, *rechts;
};
```

gelesenes Wort
wie oft gelesen?

```
BinBaum *Einlesen(string &dateiname) {
  ifstream quelle;
  quelle.open(dateiname.c_str());
  if (!quelle) return 0;
  string s;
  BinBaum *b = 0;
  while (!quelle.eof()) {
    quelle >> s;
    b = Einfuegen(s, b);
  }
  quelle.close();
  return b;
}
```

Datei öffnen

Worte einzeln auslesen und im Baum einfügen

Datei schließen + Zeiger auf Baum zurückgeben

```
BinBaum *Einfuegen(string &s, BinBaum *b) {
  if (b == 0) {
    b = new BinBaum;
    b->wort = s;
    b->anzahl = 1;
    b->links = b->rechts = 0;
    return b;
  }
  if (b->wort < s)
    b->rechts = Einfuegen(s, b->rechts);
  else if (b->wort > s)
    b->links = Einfuegen(s, b->links);
  else b->anzahl++;
  return b;
}
```

```
void Ausgabe(BinBaum *b) {
  if (b == 0) return;

  Ausgabe(b->links);
  cout << b->anzahl << " " << b->wort.c_str() << endl;
  Ausgabe(b->rechts);
}
```

Dies ist die **Inorder**-Ausgabe.

Präorder:

```
cout …;
Ausgabe(…);
Ausgabe(…);
```

Postorder:

```
Ausgabe(…);
Ausgabe(…);
cout …;
```

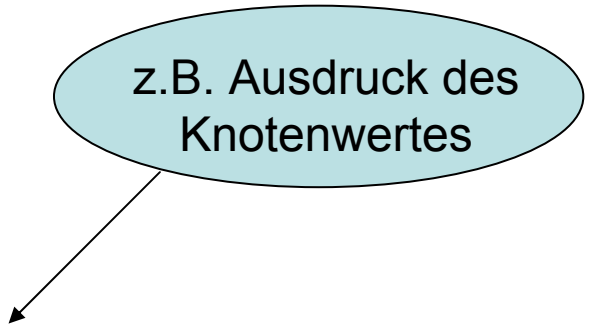
Hauptprogramm:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
  string s("quelle.txt");
  BinBaum *b = Einlesen(s);
  Ausgabe(b);
  return 0;
}
```

Durchlaufstrategien:



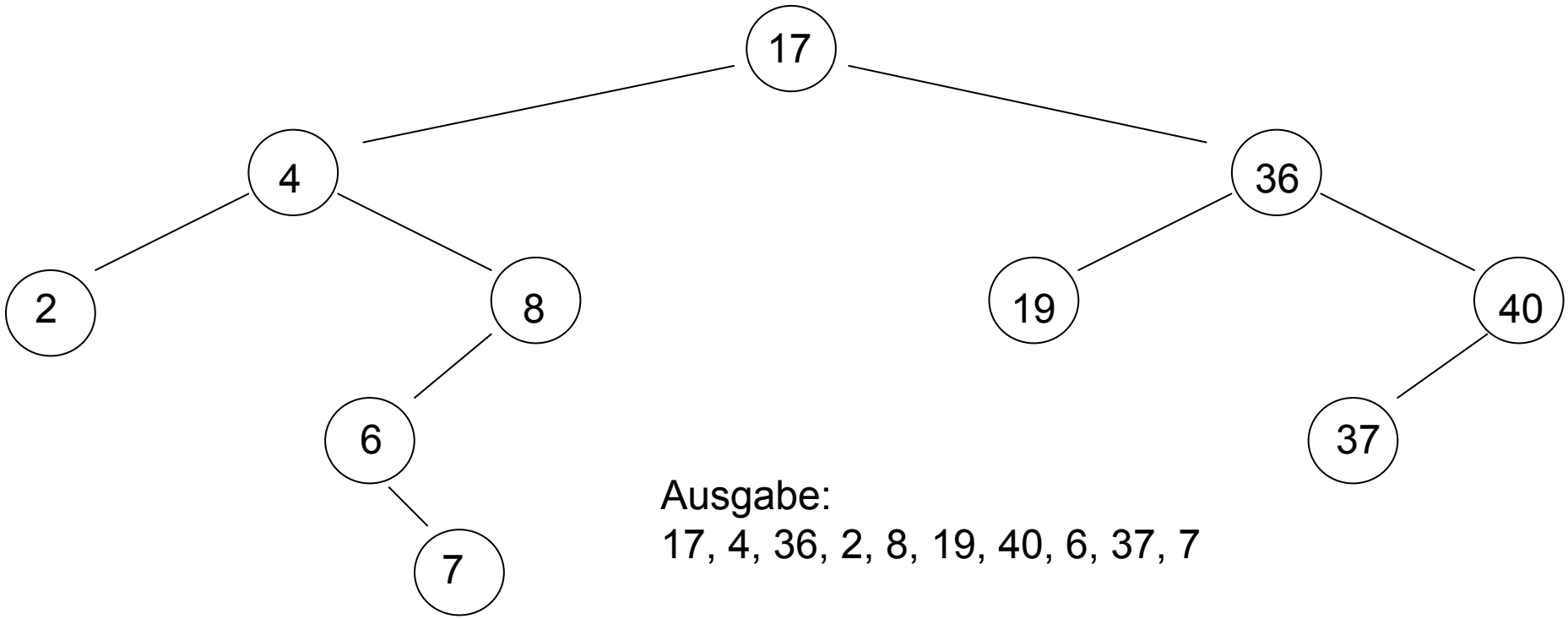
- **Tiefensuche** („depth-first search“, DFS)
  - Präorder
    Vor (*prä*) Abstieg in Unterbäume die „Knotenbehandlung“ durchführen
  - Postorder
    Nach (*post*) Abstieg in bzw. Rückkehr aus Unterbäumen die „Knotenbehandlung“ durchführen
  - Inorder
    Zwischen zwei Abstiegen „Knotenbehandlung“ durchführen

- **Breitensuche** („breadth-first search“, BFS; auch: „level search“)

  auf jeder Ebene des Baumes werden Knoten abgearbeitet, bevor in die Tiefe gegangen wird

Breitensuche

Beispiel: eingegebene Zahlenfolge 17, 4, 36, 2, 8, 40, 19, 6, 7, 37



Ausgabe:
17, 4, 36, 2, 8, 19, 40, 6, 37, 7

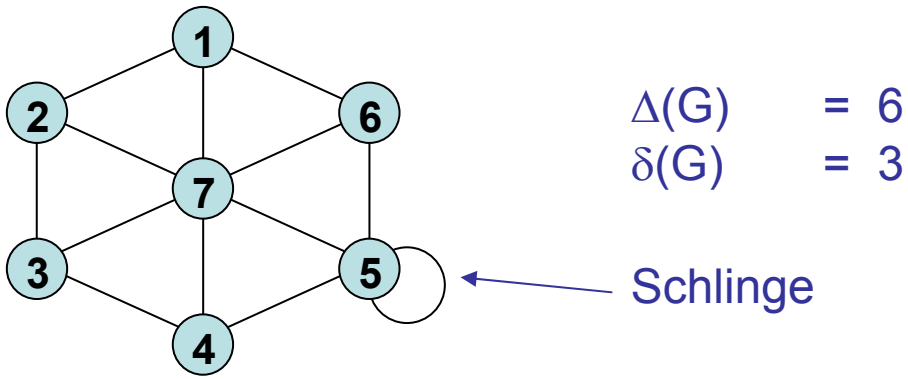
Implementierung: → Übung!

ADT Graph

- Verallgemeinerung von (binären) Bäumen
- Wichtige Struktur in der Informatik
- Zahlreiche Anwendungsmöglichkeiten
  - Modellierung von Telefonnetzen, Versorgungsnetzwerken, Straßenverkehr, …
  - Layout-Fragen bei elektrischen Schaltungen
  - Darstellung sozialer Beziehungen
  - etc.
- Viele Probleme lassen sich als Graphenprobleme „verkleiden“ und dann mit Graphalgorithmen lösen!

Definition

EIn Graph G = (V, E) besteht aus einer Menge von Knoten V („vertex, pl. vertices“) und einer Menge von Kanten E („edge, pl. edges“) mit $E \subseteq V \times V$.



$\Delta(G) = 6$

$\delta(G) = 3$

Eine Kante (u, v) heißt Schlinge (“loop“), wenn u = v.

Der Grad („degree“) eines Knotens $v \in V$ ist die Anzahl der zu ihm inzidenten Kanten: $deg(v) = | \{ (a, b) \in E : a = v \text{ oder } b = v \} |$.

Maxgrad von G ist $\Delta(G) = max \{ deg(v) : v \in V \}$

Mingrad von G ist $\delta(G) = min \{ deg(v) : v \in V \}$

Definition

Für $v_i \in V$ heißt $(v_0, v_1, v_2, \ldots, v_k)$ ein Weg oder Pfad in G,
wenn $(v_i, v_{i+1}) \in E$ für alle $i = 0, 1, \ldots, k-1$.

Die Länge eines Pfades ist die Anzahl seiner Kanten.

Ein Pfad $(v_0, v_1, v_2, \ldots, v_k)$ mit $v_0 = v_k$ wird Kreis genannt.

Distanz dist(u, v) von zwei Knoten ist die Länge des kürzesten Pfades von u nach v.

Durchmesser diam(G) eines Graphes G ist das Maximum über alle Distanzen:

$$\text{diam}(G) = \max \{ \text{dist}(u, v) : (u, v) \in V \times V \}.$$

Graph ist zusammenhängend, wenn $\forall\, u, v \in V$ mit $u \neq v$ einen Pfad gibt.

G heißt Baum gdw. G zusammenhängend und kreisfrei.

Darstellung im Computer

- Adjazenzmatrix A mit $a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$

  Problem:

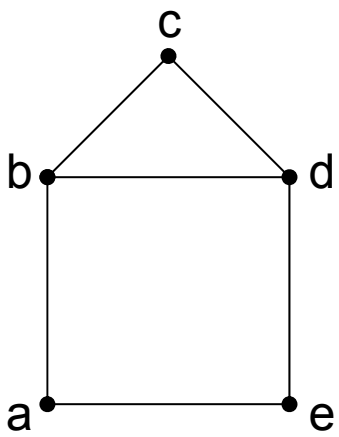
  Da $| E | \leq | V |^2 = n^2$ ist Datenstruktur ineffizient (viele Nullen) wenn | E | verschwindend klein.

- Adjazenzlisten:

  Für jeden Knoten v eine (Nachbarschafts-) Liste L(v) mit

  $$L(v) = \{ u \in V : (v, u) \in E \}$$

Beispiel



Adjazenzlisten

L(a) = (b, e)
L(b) = (a, c, d)
L(c) = (b, d)
L(d) = (b, c, e)
L(e) = (a, d)

`ADT Liste`

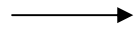
Adjazenzmatrix

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	1	0
c	0	1	0	1	0
d	0	1	1	0	1
e	1	0	0	1	0

`Array[][]`

Mögliche Funktionalität

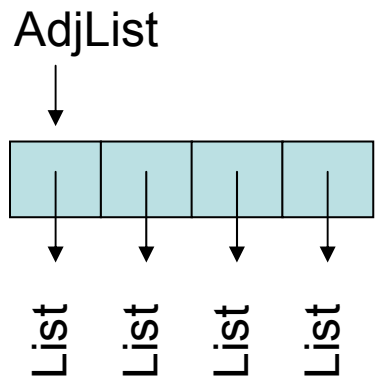
```
typedef unsigned int uint;
```



typedef Datentyp TypName;

```
Graph *createGraph(uint NoOfNodes);
void addEdge(Graph *graph, uint Node1, uint Node2);
void deleteEdge(Graph *graph, uint Node1, uint Node2);
bool hasEdge(Graph *graph, uint Node1, uint Node2);
uint noOfEdges();
uint noOfNodes();
void printGraph();
```

```
struct Graph {
  uint NoOfNodes;
  List **AdjList;      // Zeiger auf Zeiger auf Listen
};
```

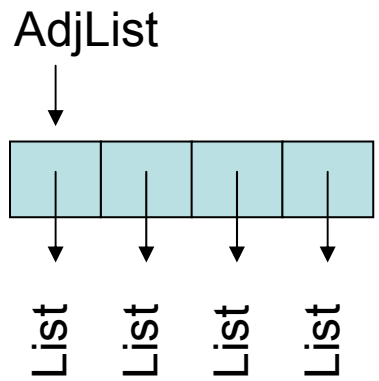



```
Graph *createGraph(uint NoOfNodes) {
  Graph *graph = new Graph;
  graph->NoOfNodes = NoOfNodes;
  graph->AdjList = new List*[NoOfNodes];
  for (uint i = 0; i < NoOfNodes; i++)
    graph->AdjList[i] = create();
  return graph;
}
```

→ Speicher reservieren

initialisieren!

```
struct Graph {
  uint NoOfNodes;
  List **AdjList;      // Zeiger auf Zeiger auf Listen
};
```



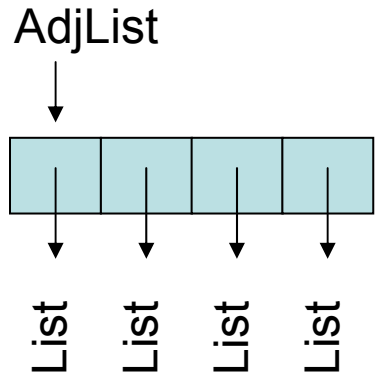

```
uint noOfNodes(Graph *graph) {
  return graph == 0 ? 0 : graph->NoOfNodes;
}

uint noOfEdges(Graph *graph) {
  if (noOfNodes(graph) == 0) return 0;
  unsigned int cnt = 0, i;
  for (i = 0; i < noOfNodes(graph); i++)
    cnt += size(graph->AdjList[i]);
  return cnt / 2;
}
```

Ineffizient!

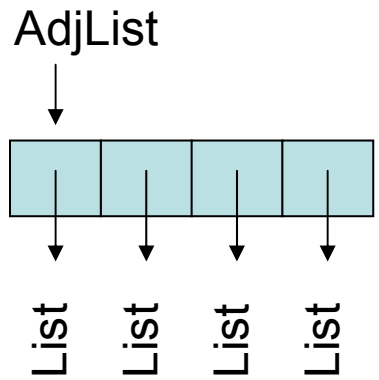
Falls häufig benutzt, dann besser Zähler **`NoOfEdges`** im **`struct Graph`**

```
struct Graph {
  uint NoOfNodes;
  List **AdjList;      // Zeiger auf Zeiger auf Listen
};
```



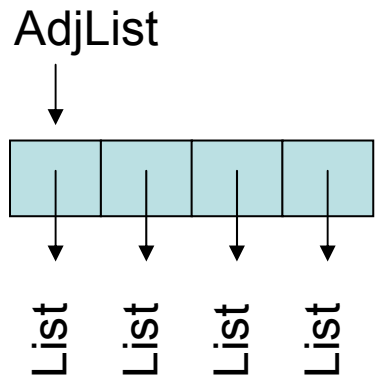

```
bool hasEdge(Graph *graph, uint Node1, uint Node2) {
  if (graph == 0) error("no graph");
  uint n = noOfNodes(graph);
  if (Node1 >= n || Node2 >= n) error("invalid node");
    return is_elem(Node2, graph->AdjList[Node1]);
}
void addEdge(Graph *graph, uint Node1, uint Node2) {
  if (!hasEdge(graph, Node1, Node2)) {
    append(Node2, graph->AdjList[Node1]);
    append(Node1, graph->AdjList[Node2]);
  }
}
```

```
struct Graph {
  uint NoOfNodes;
  List **AdjList;      // Zeiger auf Zeiger auf Listen
};
```



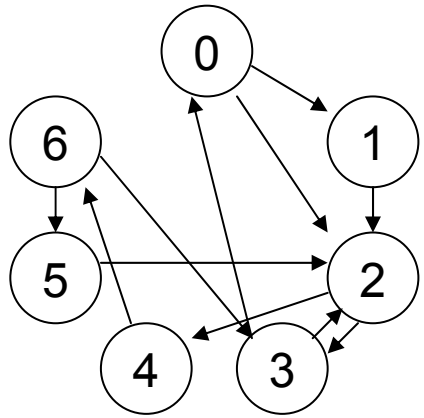

```
void printGraph(Graph *graph) {
  if (noOfNodes(graph) == 0) return;
  unsigned int i, n = noOfNodes(graph);
  for (i = 0; i < n; i++)  {
    cout << i << ": ";
    printList(graph->AdjList[i]);
  }
}
```

```
struct Graph {
  uint NoOfNodes;
  List **AdjList;      // Zeiger auf Zeiger auf Listen
};
```




```
Graph *clearGraph(Graph *graph) {
  if (graph == 0) return 0;
  unsigned int cnt = 0, i;
  for (i = 0; i < noOfNodes(graph); i++)
    clearList(graph->AdjList[i]);
  delete[] graph->AdjList;
  delete graph;
  return 0;
}
```

```
int main() {
  Graph *graph = createGraph(7);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 3, 2);
  addEdge(graph, 2, 4);
  addEdge(graph, 4, 6);
  addEdge(graph, 6, 5);
  addEdge(graph, 5, 2);
  addEdge(graph, 2, 3);
  addEdge(graph, 3, 0);
  addEdge(graph, 6, 3);
  addEdge(graph, 0, 1);
  cout << "nodes: " << noOfNodes(graph) << endl;
  cout << "edges: " << noOfEdges(graph) << endl;
  printGraph(graph);
  graph = clearGraph(graph);
  cout << "nodes: " << noOfNodes(graph) << endl;
  cout << "edges: " << noOfEdges(graph) << endl;
  return 0;
}
```




```
nodes: 7
edges: 11
0: 2 1
1: 2
2: 4 3
3: 2 0
4: 6
5: 2
6: 5 3
nodes: 0
edges: 0
```