



Wintersemester 2007/08

**Einführung in die Informatik für
Naturwissenschaftler und Ingenieure
(alias Einführung in die Programmierung)
(Vorlesung)**

Prof. Dr. Günter Rudolph

Fakultät für Informatik

Lehrstuhl für Algorithm Engineering





Kapitel 8: Elementare Datenstrukturen

Inhalt

- Definition
 - ADT Keller
 - ADT Schlange
 - ADT Liste
 - ADT Binärbaum
 - Exkurs: Einfache Dateibehandlung
 - Exkurs: Strings (C++)
 - Anwendung
 - ADT Graphen
- } heute



Exkurs: Einfache Dateibehandlung

Datei := speichert Daten in linearer Anordnung

Zwei Typen:

- **ASCII-Dateien**

- sind mit Editor les- und schreibbar
- Dateiendung („suffix“ oder „extension“) meist `.txt` oder `.asc`
- betriebssystem-spezifische Übersetzung von Zeichen bei Datentransfer zwischen Programm und externem Speicher

- **Binär-Dateien**

- werden byteweise beschrieben und gelesen
- lesen / schreiben mit Editor ist keine gute Idee
- schnellerer Datentransfer, da keine Zeichenübersetzung



Exkurs: Einfache Dateibehandlung

Hier: einfache Dateibehandlung!

- Dateien können **gelesen** oder **beschrieben** werden.
- Vor dem ersten Lesen oder Schreiben muss **Datei geöffnet** werden.
- Man kann prüfen, ob das Öffnen funktioniert hat.
- Nach dem letzten Lesen oder Schreiben muss **Datei geschlossen** werden.
- Bei zu lesenden Dateien kann gefragt werden, ob **Ende der Datei** erreicht ist.
- Beim Öffnen einer zu schreibenden Datei wird vorheriger Inhalt gelöscht!
- Man kann noch viel mehr machen ...

wir benötigen:

```
#include <fstream>           bzw.           <fstream.h>
```



Exkurs: Einfache Dateibehandlung

- Eingabe-Datei = input file

```
ifstream Quelldatei;
```



Datentyp



Bezeichner

- Öffnen der Datei:

```
Quelldatei.open(dateiName);
```

ist Kurzform von

```
Quelldatei.open(dateiName, modus);
```

wobei fehlender modus bedeutet:

ASCII-Datei,

Eingabedatei (weil ifstream)

- Ausgabe-Datei = output file

```
ofstream Zieldatei;
```



Datentyp



Bezeichner

- Öffnen der Datei:

```
Zieldatei.open(dateiName);
```

ist Kurzform von

```
Quelldatei.open(dateiName, modus);
```

wobei fehlender modus bedeutet:

ASCII-Datei,

Ausgabedatei (weil ofstream)



Exkurs: Einfache Dateibehandlung

modus:

<code>ios::binary</code>	binäre Datei
<code>ios::in</code>	öffnet für Eingabe (implizit bei ifstream)
<code>ios::out</code>	öffnet für Ausgabe (implizit bei ofstream)
<code>ios::app</code>	hängt Daten am Dateiende an
<code>ios::nocreate</code>	wenn Datei existiert, dann nicht anlegen

Warnung: teilweise Compiler-abhängig
(`nocreate` fehlt in MS VS 2003, dafür `trunc`)

Man kann diese Schalter / Flags miteinander kombinieren via:

`ios::binary | ios::app` (öffnet als binäre Datei und hängt Daten an)



Exkurs: Einfache Dateibehandlung

- **Datei öffnen**

`file.open(fileName)` bzw. `file.open(fileName, modus)`
falls Öffnen fehlschlägt, wird Nullpointer zurückgegeben

- **Datei schließen**

`file.close()`
sorgt für definierten Zustand der Datei auf Dateisystem;
bei nicht geschlossenen Dateien droht Datenverlust!

- **Ende erreicht?**

ja falls `file.eof() == true`

- **Lesen** (von ifstream)

`file.get(c);` liest ein Zeichen
`file >> x;` liest verschiedene Typen

- **Schreiben** (von ofstream)

`file.put(c);` schreibt ein Zeichen
`file << x;` schreibt verschiedene Typen



Exkurs: Einfache Dateibehandlung

Merke:

1. Auf eine geöffnete Datei darf immer nur einer zugreifen.
2. Eine geöffnete Datei belegt Ressourcen des Betriebssystems.
⇒ Deshalb Datei nicht länger als nötig geöffnet halten.
3. Eine geöffnete Datei unbekannter Länge kann solange gelesen werden, bis das Ende-Bit (end of file, EOF) gesetzt wird.
4. Der Versuch, eine nicht vorhandene Datei zu öffnen (zum Lesen) oder eine schreibgeschützte Datei zu öffnen (zum Schreiben), führt zu einem Nullpointer.
⇒ Das muss überprüft werden, sonst Absturz bei weiterer Verwendung!
5. Dateieingabe und -ausgabe (input/output, I/O) ist sehr langsam im Vergleich zu den Rechenoperationen.
⇒ I/O Operationen minimieren.

"The fastest I/O is no I/O."

Nils-Peter Nelson, Bell Labs



Exkurs: Einfache Dateibehandlung

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

using namespace std;

int main() {                                // zeichenweise kopieren
    ifstream Quelldatei;
    ofstream Zieldatei;

    Quelldatei.open("quelle.txt");
    if (!Quelldatei.is_open()) {
        cerr << "konnte Datei nicht zum Lesen öffnen\n";
        exit(1);
    }
    Zieldatei.open("ziel.txt");
    if (!Zieldatei.is_open()) {
        cerr << "konnte Datei nicht zum Schreiben öffnen\n";
        exit(1);
    }
}
```



Exkurs: Einfache Dateibehandlung

```
while (!Quelldatei.eof()) {  
    char c;  
    Quelldatei.get(c);  
    Zieldatei.put(c);  
}  
  
Quelldatei.close();  
Zieldatei.close();  
}
```

offene
Datei



Start



aktuelle Position



eof() == true



Exkurs: C++ Strings

Bisher:

Zeichenketten wie `char str[20];`

- Relikt aus C-Programmierung!
- bei größeren Programmen mühevoll, lästig, ...
- ... und insgesamt **fehlerträchtig!**

Jetzt:

Zeichenketten aus C++

- sehr angenehm zu verwenden (keine 0 am Ende, variable Größe, ...)
- eingebaute (umfangreiche) Funktionalität

wie benötigen: `#include <string>` und `using namespace std;`



Datendefinition / Initialisierung

```
string s1; // leerer String
string s2 = "xyz"; // initialisieren mit C-String
string s3 = s2; // vollständige Kopie!
string s4("abc"); // initialisieren mit C-String
string s5(s4); // initialisieren mit C++-String
string s6(10, '*'); // ergibt String aus 10 mal *
string s7(1, 'x'); // initialisieren mit einem char
string sx('x'); // FEHLER!
string s8(""); // leerer String
```



Exkurs: C++ Strings

Eingebaute Funktionen

- Konvertierung C++-String nach C-String via `c_str()`

```
const char *Cstr = s2.c_str();
```

- Stringlänge `length()`

```
cout << s2.length();
```

- Index von Teilstring finden

```
int pos = s2.find("yz");
```

- Strings addieren

```
s1 = s2 + s3;  
s4 = s2 + "hello";  
s5 += s4;
```

- Strings vergleichen

```
if (s1 == s2) s3 += s2;  
if (s3 < s8) flag = true;
```

- `substr()`,
`replace()`,
`erase()`,
...



Kapitel 8: Elementare Datenstrukturen

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.



Kapitel 8: Elementare Datenstrukturen

```
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.is_open()) 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



Kapitel 8: Elementare Datenstrukturen

```
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;
}
```




Kapitel 8: Elementare Datenstrukturen

```
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 ...;
```



Kapitel 8: Elementare Datenstrukturen

Hauptprogramm:

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

using namespace std;

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



Kapitel 8: Elementare Datenstrukturen

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

z.B. Ausdruck des
Knotenwertes

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

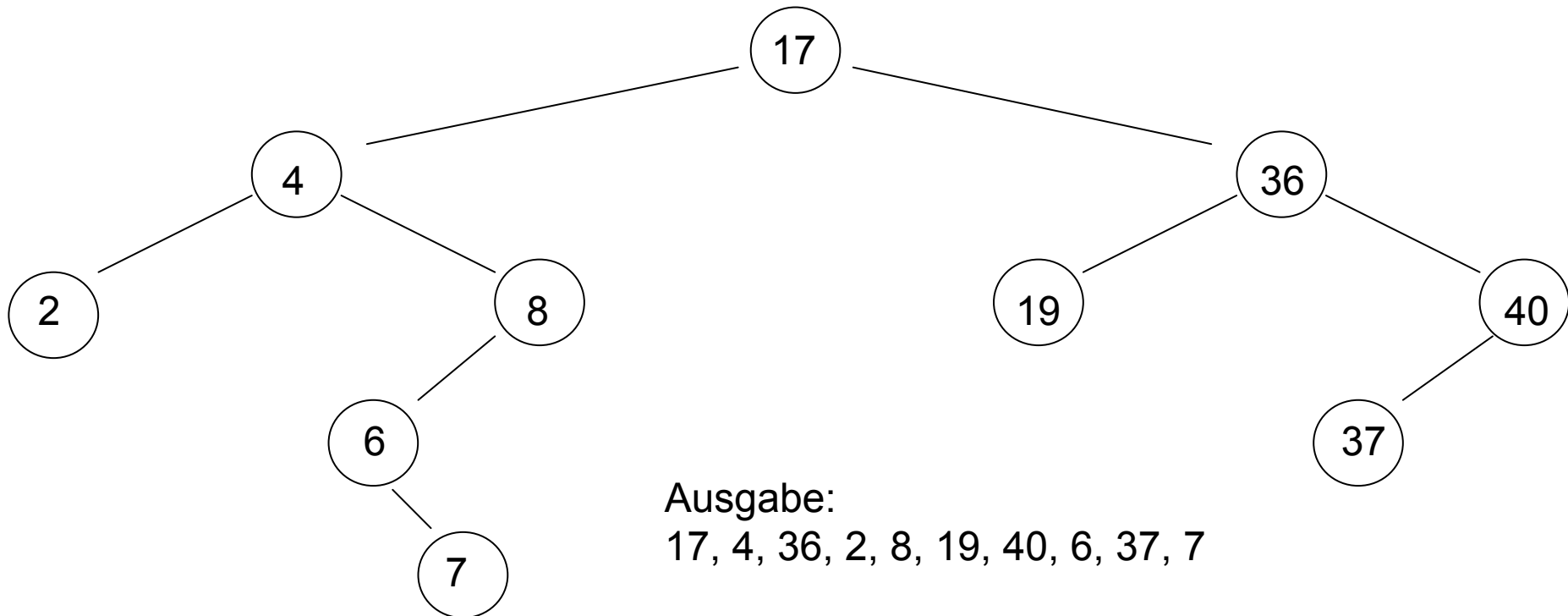
auf jeder Ebene des Baumes werden Knoten abgearbeitet,
bevor in die Tiefe gegangen wird



Kapitel 8: Elementare Datenstrukturen

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: → Praktikum!



Kapitel 8: Elementare Datenstrukturen

ADT Graph

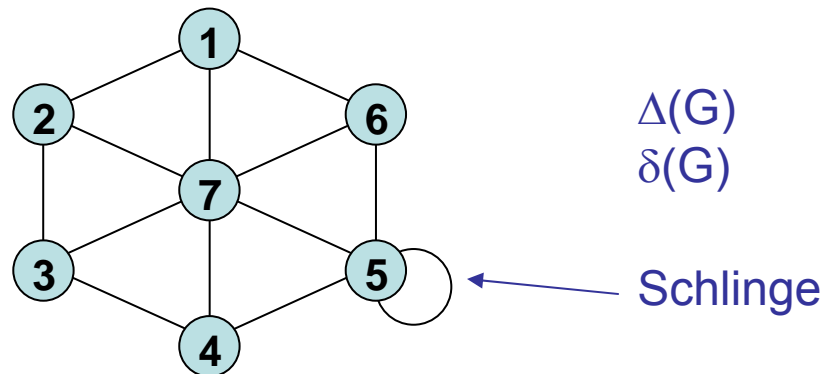
- Verallgemeinerung von (binären) Bäumen
- Wichtige Struktur in der Informatik
- Zahlreiche Anwendungsmöglichkeiten
 - Modellierung von Telefonnetzen, Versorgungsnetzwerken, Straßenverkehr, ...
 - Layout-Fragen bei elektrischen Schaltungen
 - Darstellung sozialer Beziehungen
 - etc.
- Viele Probleme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



Kapitel 8: Elementare Datenstrukturen

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 $E \subseteq V \times V$.



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



Kapitel 8: Elementare Datenstrukturen

Definition

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

Die Länge eines Pfades ist die Anzahl seiner Kanten.

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

Distanz $\text{dist}(u, v)$ von zwei Knoten ist die Länge des kürzesten Pfades von u nach v .

Durchmesser $\text{diam}(G)$ 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.



Kapitel 8: Elementare Datenstrukturen

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:

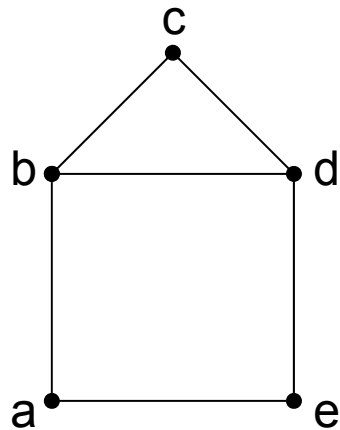
Für jeden Knoten v eine (Nachbarschafts-) Liste $L(v)$ mit

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



Kapitel 8: Elementare Datenstrukturen

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[][]



Kapitel 8: Elementare Datenstrukturen

Mögliche Funktionalität

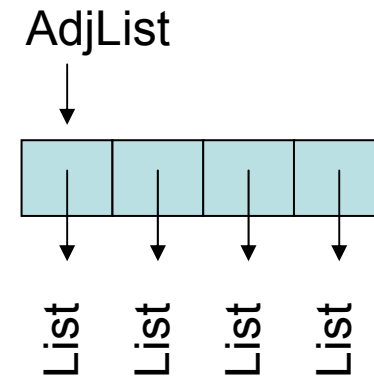
```
typedef unsigned int uint;    ———>  typedef Datentyp TypName;
```

```
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();
```



Kapitel 8: Elementare Datenstrukturen

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



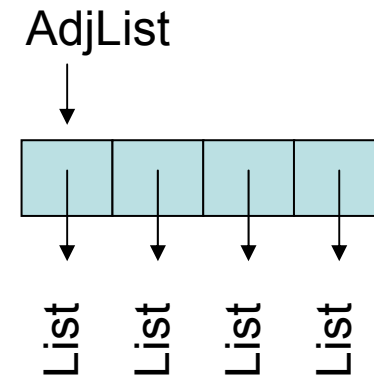
```
Graph *createGraph(uint NoOfNodes) {  
    Graph *graph = new Graph;  
    graph->NoOfNodes = NoOfNodes;  
    graph->AdjList = new List*[NoOfNodes];  
    for (uint i = 0; i < NoOfNodes; i++)  
        graph->AdjList[i] = create();  
    return graph;  
}
```

→ Speicher reservieren
} initialisieren!



Kapitel 8: Elementare Datenstrukturen

```
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;  
}
```

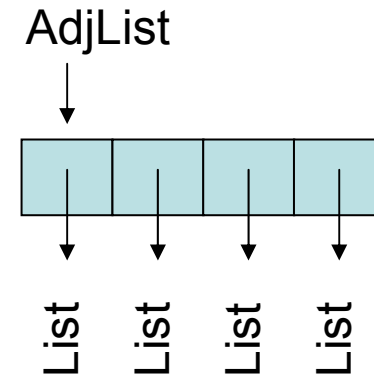
Ineffizient!

Falls häufig
benutzt, dann
besser Zähler
NoOfEdges im
struct Graph



Kapitel 8: Elementare Datenstrukturen

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

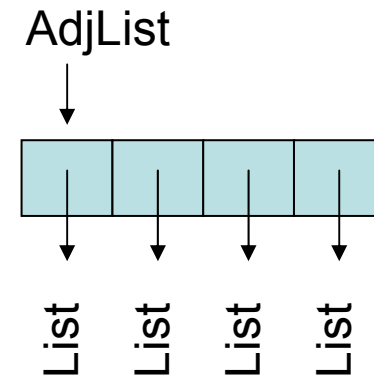


```
bool hasEdge(Graph *graph, uint Node1, uint Node2) {  
    if (graph == 0) error("no graph");  
    uint n = noOfNodes(graph);  
    if (Node1 >= n || Node2 >= n) error("invalid node");  
    return is_elem(Node2, graph->AdjList[Node1]);  
}  
void addEdge(Graph *graph, uint Node1, uint Node2) {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



Kapitel 8: Elementare Datenstrukturen

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

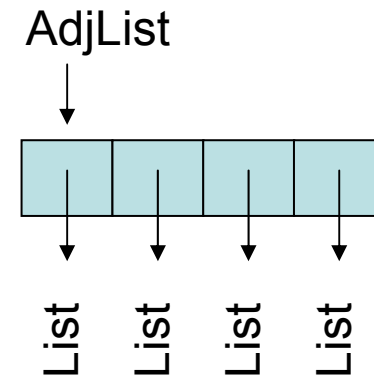


```
void printGraph(Graph *graph) {  
    if (noOfNodes(graph) == 0) return;  
    unsigned int i, n = noOfNodes(graph);  
    for (i = 0; i < n; i++) {  
        cout << i << ": ";  
        printList(graph->AdjList[i]);  
    }  
}
```



Kapitel 8: Elementare Datenstrukturen

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

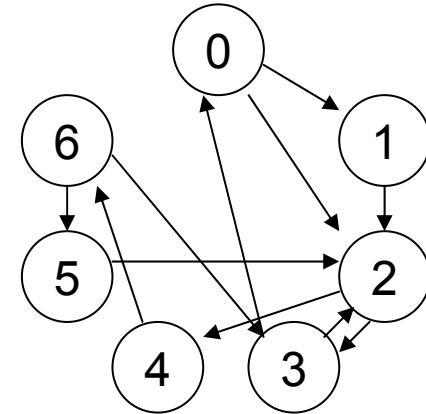


```
Graph *clearGraph(Graph *graph) {  
    if (graph == 0) return 0;  
    unsigned int cnt = 0, i;  
    for (i = 0; i < noOfNodes(graph); i++)  
        clearList(graph->AdjList[i]);  
    delete[] graph->AdjList;  
    delete graph;  
    return 0;  
}
```



Kapitel 8: Elementare Datenstrukturen

```
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
```