Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

10. Vorlesungswoche

Dynamische Datenstrukturen:

Listen, Bäume, Graphen, Schlangen, Keller, Mengen

Unterlagen:

Echtle, Goedicke; Einführung in die objektorientierte Programmierung mit Java, dpunkt-Verlag.

Doberkat, Dissmann; Einführung in die objektorientierte Programmierung mit Java, Oldenbourg-Verlag, 2. Auflage.

Folien nach V.Gruhn, Vorlesung Programmierung WS 99/00

Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Überblick

- Dynamische Datenstrukturen:
  - Strukturen, die je nach Bedarf und damit dynamisch wachsen und schrumpfen können (Unterschied zu Felder/Arrays!).
- Grundidee:
  - Dynamische Datenstrukturen bilden Mengen mit typischen Operationen ab
  - Einzelne Elemente speichern die zu speichernden / zu verarbeitenden Daten
  - Einzelne Elemente werden durch dyn. Datenstrukturen verknüpft

  also: Trennung von Datenverwaltung & Speicherung.
- Art der Elemente ist problemabhängig, variiert je nach Anwendung
- Für die Verknüpfung existieren typische Muster

  => dynamische Datenstrukturen: Liste, Baum, Graph, ...

  mit zugehörigen Zugriffsmethoden
- Objektorientierte Sicht: dynamische Datenstrukturen durch die Art der Verknüpfung der Elemente und die Zugriffsmethoden charakterisiert.

Wichtige dynamische Datenstrukturen

- Listen, lineare Listen, doppelt verkettete Listen
- Bäume, binäre Bäume, binäre Suchbäume
- Graphen, gerichtete Graphen, ungerichtete Graphen
- Stack, Schlangen
- Mengen

Fragen zur Organisation der Datenstrukturen:

– Wie wird eine Instanz der Struktur initialisiert, Daten eingefügt, modifiziert, entfernt ?
– Wie wird in den Strukturen navigiert ?
– Wie werden einzelne Werte in einer Struktur wiedergefunden ?
– Wie werden alle in einer Struktur abgelegten Werte besucht ?

Aus Sicht einer Menge: Vereinigung, Schnittmenge (mit einelementigen Mengen)

Allgemeines zu Graphen

Dynamische Strukturen im Vergleich:

- Listen: eindimensionale Verkettungen von Informationen:
  - Jeder Knoten einer Liste hat einen oder keinen Nachfolger.
- Binäre Bäume: zweidimensionale Informationsverkettung:
  - Jeder Knoten ist mit höchstens zwei Söhnen verbunden.
- Graphen: n-dimensionale Verkettung:
  - für jeden Knoten angegeben, in welcher Weise er mit anderen Knoten (gerichtet oder ungerichtet) verbunden ist.
  - falls n endlich ist, spricht man von beschränktem Eingangs-/Ausgangsgrad an Knoten

Allgemeines zu Graphen

- originäre Anwendung:
  - Darstellung räumlicher Beziehungen,
  - Bäume unzureichend, weil Beziehungen zwischen allen Knoten möglich.
- häufig: knoten/kanten-annotierte Graphen
  - weitere Informationen/Attribute an Kanten: Kosten, Entfernungen, etc..
- ungerichteter Graph:
  - Wenn die Beziehung zwischen zwei Knoten unabhängig von der Richtung ist (z.B. Erreichbarkeit zwischen Orten), dann reichen ungerichtete Kanten.
  - Kanten als symmetrische Relation über die Knotenmengen
- gerichteter Graph:
  - Wenn die Beziehung von der Richtung abhängt (eine Stellung im Schach lässt sich per Zug aus einer anderen erreichen), dann braucht man gerichtete Kanten.
  - Kanten als Relation über die Knotenmengen

Allgemeines zu Graphen

Typische Methoden:

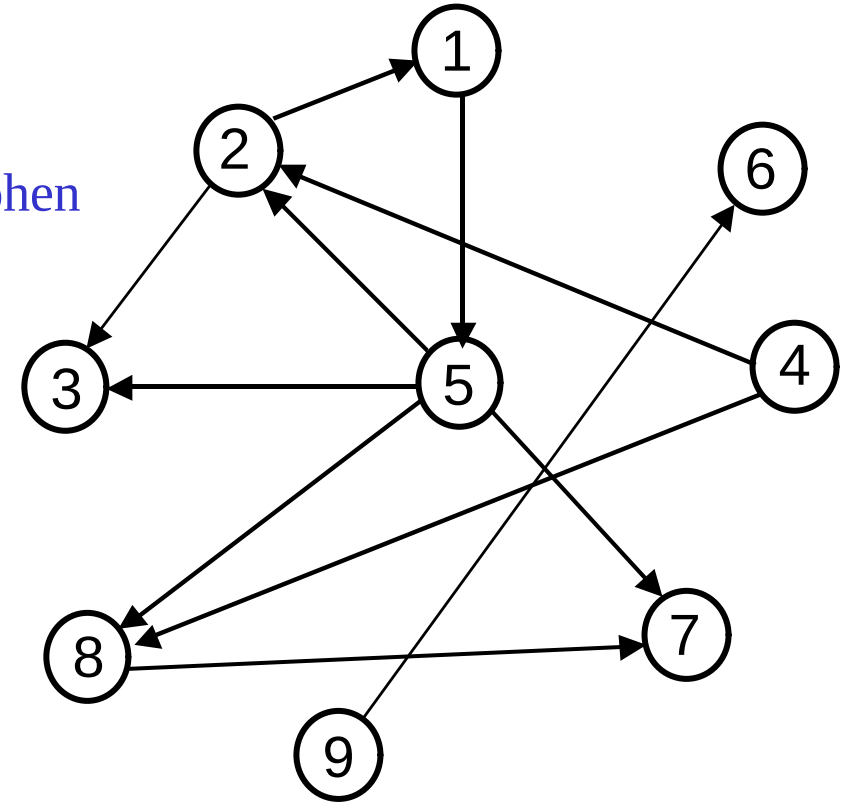
- das Erzeugen eines neuen Graphen
- das Einfügen oder Löschen eines Knotens
- das Einfügen oder Löschen einer Kante
- den Test, ob ein Knoten vorhanden ist
- den Test, ob eine Kante vorhanden ist

Definition Graph

G = (V, E) heißt *gerichteter Graph* mit *Knotenmenge V* und *Kantenmenge E*, falls gilt:

- *V* (englisch: vertex, vertices) ist eine endliche Menge und
- *E* (englisch: edges) $\subseteq V \times V$

Beispiel eines gerichteten Graphen



Definition Pfad

Sei $G = (V, E)$ ein gerichteter Graph, so besteht ein **Pfad/Weg** aus Knoten $(p_0, p_1, \ldots, p_k)$ von p_0 nach p_k für $k > 0$ aus Knoten $p_0, \ldots, p_k \in V$, so dass jedes Paar aufeinanderfolgender Knoten (p_i, p_{i+1}) eine Kante bildet. Dies gilt für $0 \leq i \leq k\text{-}1$.

- **einfacher Weg:** Weg, auf dem kein Knoten doppelt vorkommt
- **Zyklus:** einfacher Weg von einem Knoten zu sich selbst.
- **Semiweg:** Folge von Knoten $(p_0, p_1, \ldots, p_k)$, wobei jeweils zwischen (p_i, p_{i+1}) oder (p_{i+1}, p_i) eine Kante besteht.
  - Semiweg entspricht Pfad auf einem zugehörigen ungerichteten Graphen.

Definition Zusammenhang

- Ein gerichteter Graph heißt **stark zusammenhängend**, wenn es zwischen je zwei Knoten des Graphen einen Pfad gibt.
- Ein gerichteter Graph heißt **schwach zusammenhängend**, wenn es zwischen zwei Knoten immer einen **Semiweg** gibt.
- Ein zusammenhängender, zyklenfreier Graph ist ein Baum.

Definition Zusammenhangskomponente

Ein Teilgraph heißt **Zusammenhangskomponente**, wenn er bzgl. der Zusammenhangseigenschaft (stark oder schwach) maximal ist, d.h. der Teilgraph kann nicht durch einen oder mehrere Knoten und/oder eine oder mehrere Kanten des Graphen erweitert werden, ohne diese Eigenschaft zu verlieren.

Teilgraph: R=(V,E´) ist Teilgraph von G=(V,E) falls, $E' \subseteq E$

Spannbaum: Falls G zusammenhängend und R zusammenhängend und zyklenfrei, dann ist R Spannbaum (erzeugender Baum)

- Von Interesse sind oft minimale Spannbäume
- Schema zur Erzeugung eines Spannbaumes:
  solange es einen Zyklus gibt, entferne eine Kante aus diesem Zyklus

Speicherdarstellung von Graphen

- durch Adjazenzlisten
- durch Adjazenzmatrizen

Adjazenz von Knoten

Ist $n \in V$ ein Knoten im gerichteten Graphen (V, E), so heißt $\{p \in V; (n, p) \in E\}$ die *Adjazenzliste zum Knoten n.*

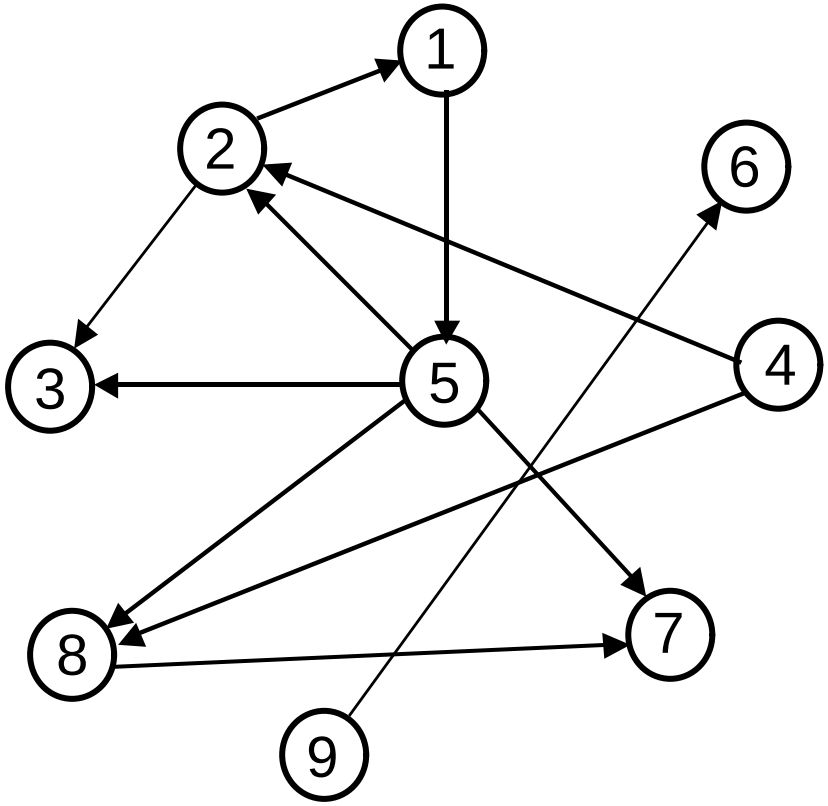
Eine Alternative zu dieser Darstellung ist die Darstellung über Boolesche Matrizen:
Man definiere eine Matrix $(a_{k,l})_{k,l \in V}$ und setze:

$$a_{k,l} := ((k,l) \in E)$$

Bei kantenbewerteten Graphen können natürlich auch die Werte an einer Kante in die Matrix eingetragen werden.

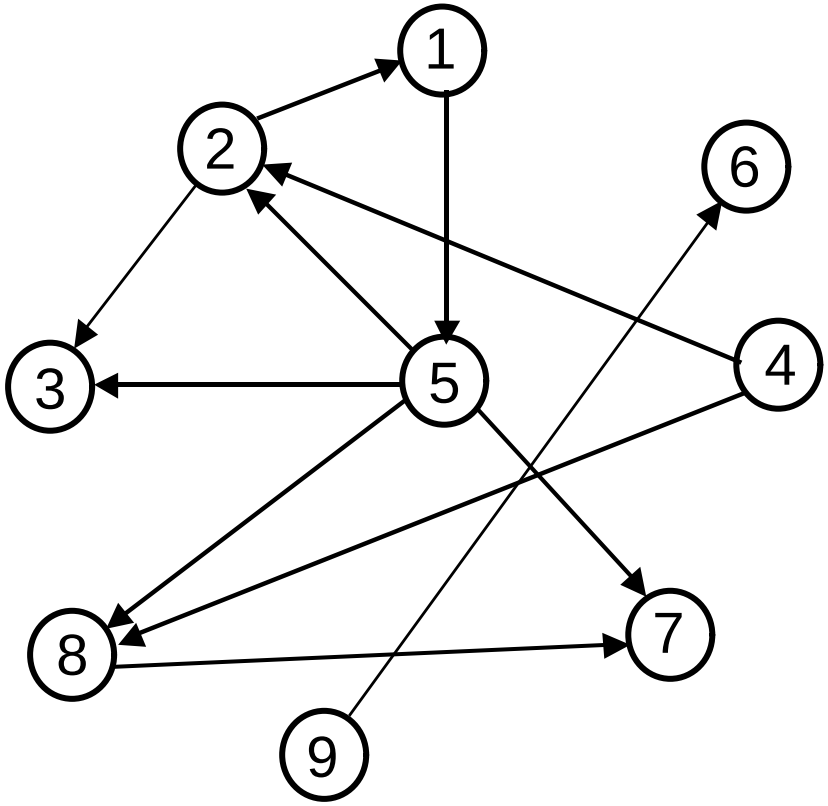
- Übliche Interpretation: 0 = keine Kante
- bei mehreren Kanten zwischen Knoten k, l sind entweder mehrere Matrizen oder eine sinnvolle arithmetische Operation zur Aggregierung erforderlich, üblich: Addition der Kantengewichte (mögliche Alternativen: Max, Min,...)

Beispiel Adjazenzliste



Knoten	Adjazenzliste
1	5
2	1, 3
3	(leer)
4	2, 8
5	2, 3, 7, 8
6	
7	(leer)
8	7
9	6

Beispiel Adjazenzmatrix



Adjazenzmatrix

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

Adjazenz von Knoten

Adjazenzmatrizen:

- Platzverbrauch: n^2 Speicherplätze, unabhängig von der Kantenzahl
- Methoden: direkter Zugriff auf Einträge: Einfügen/Suchen/Ändern/Löschen in O(1) durch indizierten Zugriff auf zweidimensionales Array

Adjazenzlisten:

- Platzverbrauch: n+k Speicherplätze bei k Kanten
- Methoden: bei Speicherung mit linearen Listen, Einfügen/Suchen/Ändern/Löschen in O(x) bei x Kanten für einen Knoten.

Deshalb folgende Daumenregel:

- Adjazenzlisten für Graphen mit wenig Kanten pro Knoten.
- Adjazenzmatrizen für dichte Graphen geeignet.

Randbemerkung:

In der Literatur auch Inzidenzmatrizen: boolesche Matrizen mit Knoten als Zeilen, Kanten als Spalten, Eintrag M[i,j]=1 zeigt an, dass Knoten i an Kanten j liegt.

Bewertung: Inzidenzmatrizen eher unpraktisch (Platzbedarf, gerichtete Kanten, bewertete Kanten, ...)

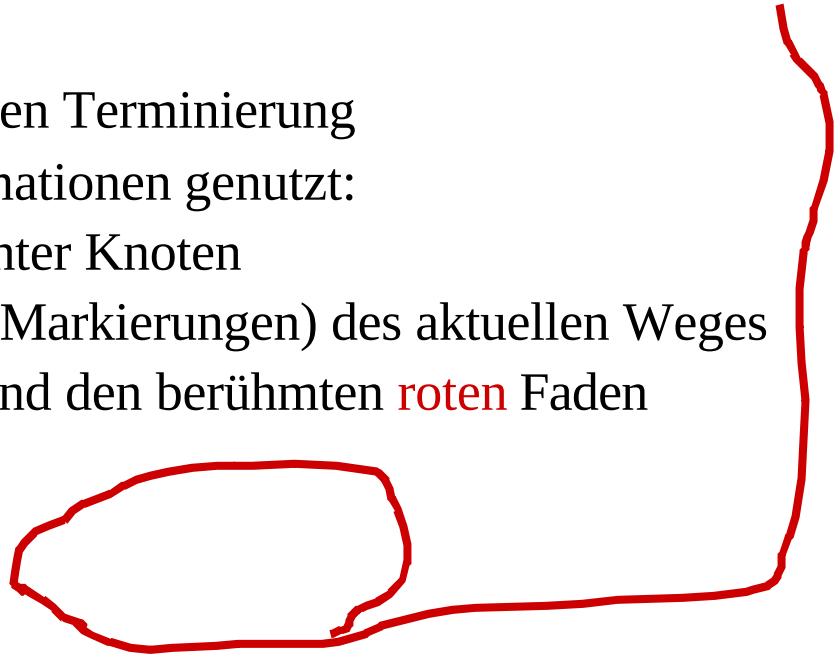
Einige interessante Algorithmen für Graphen

- Traversierungen
  - Tiefendurchlauf
  - Breitendurchlauf
- transitive Hülle
- kürzeste Wege
- starke Zusammenhangskomponenten

- aber nicht alle Fragestellungen sind einfach zu lösen
  - Travelling Salesman Problem, Routenplanung, etc.

Traversierung

- Traversieren:
  - Durchwandern aller Knoten (bzw Kanten) eines Graphen
  - interessant sind nur zusammenhängende Graphen
    - sonst zerfällt Problem in k unabhängige Teilprobleme
- Ähnliche Ansätze wie bei Bäumen
  - Tiefensuche, Breitensuche
  - Besonderheit: Zyklen gefährden Terminierung
  - daher typischerweise 2 Informationen genutzt:
    - Markierung bereits besuchter Knoten
    - Merken (evtl anhand von Markierungen) des aktuellen Weges
      - erinnert an Ariadne und den berühmten roten Faden



Traversierung von Graphen

- Tiefensuche (depth first search, dfs)

```
Algorithmusschema für rekursiven Tiefendurchlauf
void dfs(int vertex) {
  if (!markiert(vertex)) {
     markiere(vertex);
     für alle Nachfolgeknoten v: dfs(v) ;
}}
```

nur unmmarkierte Knoten werden bearbeitet, markiert => je Knoten 1 Bearbeitung, je Kante 1 Funktionsaufruf

- Breitensuche (breadth first search, bfs)
  - Grundalgorithmus wie bei Bäumen
  - Algorithmus arbeitet mit Warteschlange als Hilfsspeicher
  - Knoten werden einzeln zur Bearbeitung der Warteschlange von vorn entnommen
  - neu entdeckte (unmarkierte) Knoten werden markiert und an das Ende der Warteschlange angefügt

Transitive Hülle

- transitive Hülle einer zweistelligen Relation R:
  t(R) ist die kleinste transitive Relation, die R enthält.
- beantwortet Fragen nach der Existenz von Wegen zwischen Knoten
- Berechnung nach Floyd Warshall (für Adjazenzmatrix A)

```
void WarshallAlgo( boolean [][] A, int Max) {
for (int y=0; y < Max ; y++) {                  // Wege über y
   for (int x=0; x < Max ; x++) {               // beginnend in x
         if (A[x][y]) {
            for (int z=0; z < Max ; z++){
                  if (A[y][z])                  // endend in z
                     A[x][z] = 1                // nun auch direkt
            }
   }              // Aufwand O(n^3)
}}                // Korrektheit: Induktion über äußere Schleife
```

Kürzeste Wege

- Kürzester Weg: Weg bei dem die Summe der Kantengewichte minimal wird
- Verallgemeinerung von Warshalls Algorithmus
  - statt A[x][z]=1, setze A[x][z]=min(A[x][z], A[x][y]+ A[y][z])
  - Algorithmus als Floyd Algorithmus bekannt
- Falls 1 kürzester Weg (und nicht alle) interessant: Dijkstras Algorithmus, Grundidee: kürzester Weg besteht aus kürzesten Teilwegen,

  bei intelligenter Bestimmung minimaler Kanten $O(n^2)$

gegeben: Graph G=(V,E)

gesucht: kürzester Weg von u nach v

Algorithmusskizze

Sei S Menge aller Knoten für die kürzester Entfernung zu u bekannt, initial S={u}.

solange $v \notin S$

- wähle Kante (x,y), $x \in S$, $y \notin S$, mit minimalen Kosten,

  d.h. kosten(u,x)+kosten(x,y) von allen möglichen Kanten (x,y) minimal.
- $S = S \cup \{y\}$

Starke Zusammenhangskomponenten

- Algorithmus nach Tarjan, maximale starke ZKs für gerichtete Graphen
- Grundidee
  - Tiefensuche mit fortlaufender Numerierung der Knotenmenge und Bewertung eines Knotens mit dem Minimum erreichbarer Nummern über T*B Wege.
  - Vergabe der Nummer bei erstmaligem Erreichen
  - Festlegen der Bewertung nach Betrachtung aller Nachfolger als Minimum von Wegen, die aus Tree Kanten und maximal einer Back Kante bestehen.
  - dabei entsteht Partitionierung der Kantenmenge in
    - Tree Kanten: Kanten, die einen Spannbaum erzeugen, also zu Knoten führen, die erstmalig erreicht werden und eine Nummer erhalten
    - Back Kanten: Kanten, die Zyklen bilden und im Spannbaum aufwärts führen, also zu einem Knoten führen, der auf dem aktuellen Pfad liegt
    - Cross Kanten: Kanten, die vom aktuellen Teilbaum zu einem Teilbaum führen, der bereits komplett bearbeitet wurde, also zu Knoten führen, die bekannt sind und nicht mehr auf dem aktuellen Pfad liegen
    - Forward Kanten: Kanten, die im aktuellen Teilbaum einige Ebenen überspringen und Wege aus Tree Kanten abkürzen
- Details zum Algorithmus und zur Argumentation in der Literatur
- faszinierend: Aufwand: O(|V|+|E|)

  => es existieren sehr geschickte und sehr effiziente Graphalgorithmen !!!!

Travelling Salesman Problem

aber nicht alle Graphprobleme sind einfach zu lösen:

„Berechne für gerichteten Graphen G=(V,E) den kürzesten Weg von u nach v, in dem alle anderen Knoten mindestens einmal (genau einmal) besucht werden.“

- Motivation: Tourenplanung für Handlungsreisenden
- Ursprung: u=v und Knoten werden genau einmal besucht
- Algorithmen mit Laufzeit $O((|V|+|E|)^k)$, der das Problem vollständig löst, ist nicht bekannt, und Existenz ist leider auch nicht zu erwarten ...
- Algorithmusskizze
  - zähle alle geeigneten Wege durch einen Tiefensuchalgorithmus auf
  - merke den günstigsten Weg
- Tiefensuche wird als Backtracking realisiert und nach Besuch des Knotens und Betrachten aller Wege von diesem Knoten aus, wird die Markierung wieder zurückgesetzt.
- Laufzeit des Verfahrens $O(2^{|V|})$
- Grundproblem: Menge der Wege ist exponentiell in der Anzahl Knoten, z.B. bei einem Graphen mit 2 Ausgangskanten je Knoten 2^N Wege der Länge N

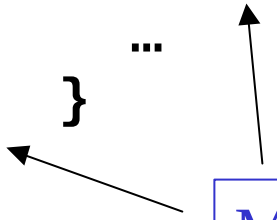
Implementierungsbeispiel: Klassen Knoten, ..., Graph

```
class Knoten {
   private int name;
   private KnotenListe
   adjazenz;
   private Knoten nächster;
   …
}

class KnotenListe {
   private KnotenElem kopf,
   fuß;
   …
}

class KnotenElem {
   Knoten verweis;
   KnotenElem weiter;
   …
}
```

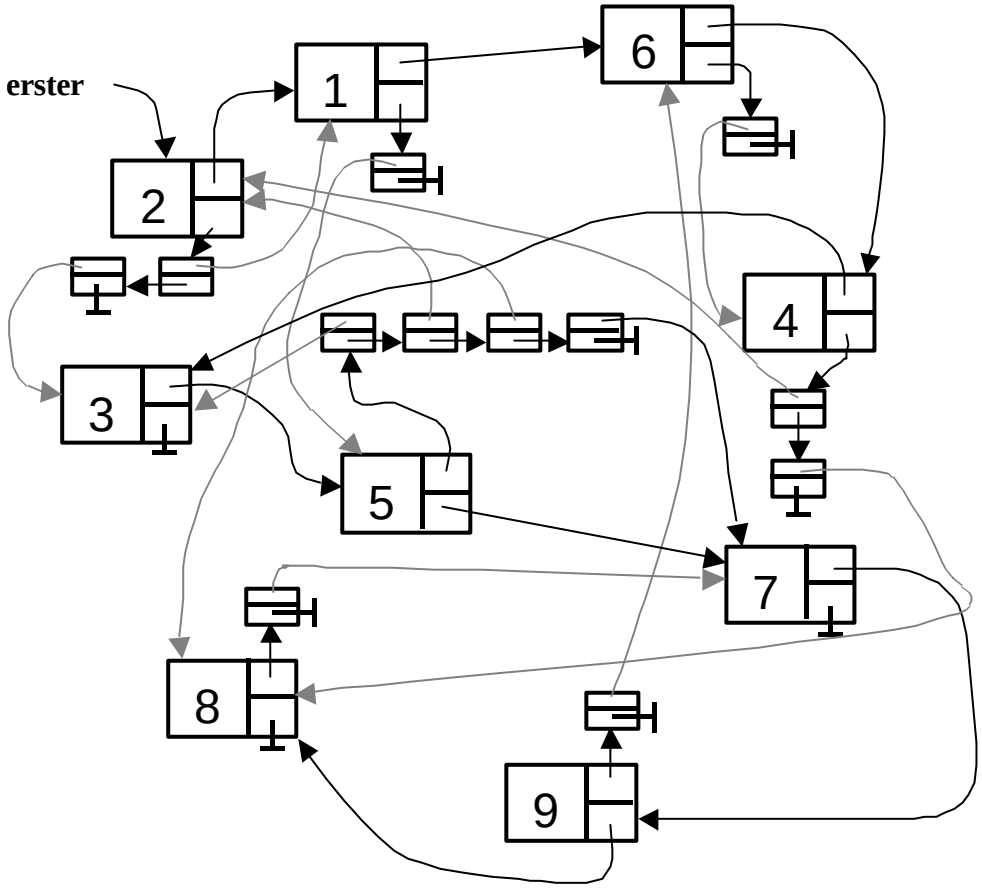
```
class Graph {
   private Knoten erster;
   …
}
```



Menge der Knoten V
als lineare Liste realisiert

Menge der Ausgangskanten
als Adjazenzliste an jedem
Knoten mit linearer Liste realisiert

Listendarstellung gerichteter Graphen



Realisierung der Graph-Methoden

- Erzeugen eines neuen Graphen
  - erfolgt durch den Aufruf eines Konstruktors.
- Hinzufügen eines Knotens
  - ist eine Listenoperation auf der Liste der Knoten.
- Entfernen eines Knotens
  - ist eine Listenoperation auf der Liste der Knoten und auf all den Adjazenzlisten, die einen Zeiger auf den zu entfernenden Knoten besitzen. Aufwand!
- Hinzufügen einer Kante
  - ist eine Operation auf den Adjazenzlisten.
- Test, ob ein Knoten vorhanden ist,
  - ist eine Listenoperation auf der Liste der Knoten.
- Test, ob eine Kante vorhanden ist,
  - ist zunächst eine Listenoperation auf der Liste aller Knoten
    - Ausgangsknoten der Kante finden
  - anschließend Listenoperation auf der entsprechenden Adjazenzliste.

Die Klasse Graph

```
class Graph {
      private Knoten erster;
      void Graph() { … }
      void FügeKnotenEin(int kno) { … }
      void EntferneKnoten(int kno) { … }
      void FügeKanteEin(int start, int ende) { … }
      boolean IstKnotenDa(int kno) { … }
      boolean IstKanteDa(int kno1, int kno2) { … }
  }
```

Durchlaufen mit einmaligem Besuchen

- Beispielanwendung: Ausdrucken aller Knotennamen
- Einfache Ausgabestrategie:
  - Durchlaufen der Knotenliste, allerdings verliert man dann jeden Aufschluss über die Graphenstruktur.
- Deshalb Tiefendurchlauf :
  - Hierdurch werden zusammenhängende Teilgraphen auch zusammenhängend ausgegeben.
- Graphen haben keine vergleichbar reguläre Struktur wie Bäume,
  - Problem: Zyklen gefährden Terminierung, müssen erkannt werden.
  - Durchlaufstrategien wie bei Bäumen müssen angepasst werden.
  - Idee bereits diskutiert: Markierung

Durchlaufen mit einmaligem Besuchen

Klasse Knoten wird deshalb um Attribut **warDa** erweitert.

- warDa dient zur Speicherung einer Markierung, unseres roten Fadens

```
class Knoten {
      private boolean warDa;
      void SetzeWarDa(boolean wd) { warDa = wd; }
      boolean GibWarDa() { return warDa; }
      … }
```

Problem:

- Was passiert, wenn mehrfach ausgedruckt werden soll ?
- Werden warDa Einträge zurückgesetzt ? (Aufwand!)

- Idee 1: Invertieren der Werte.
  - Das Einfügen muss dann aber entsprechend arbeiten und die Belegung der warDa-Attribute berücksichtigen.
- Idee 2: Integer statt Boolescher Werte, Inkrementieren der Werte

Durchlaufen mit einmaligem Besuchen

Anmerkungen zum Ablauf:

- Erst wenn in einem Teilgraph alle Knoten besucht sind, wird über **`nächster`** der nächste Knoten aufgesucht.
- Wir versuchen von einem Knoten alle erreichbaren Knoten zu finden. Dabei steigen wir erst hinab und erst wieder hinauf (zum Vorgängerknoten), wenn nichts mehr gefunden wird.
- Das Durchlaufen einer Adjazenzliste wird durch den rekursiven Aufruf von **`BesucheTeil`** für die in der Adjazenzliste vorkommenden Knoten unterbrochen.
- Die Methoden **`InitIteration`** und **`GeheWeiter`** sorgen dafür, dass die Adjazenzliste dennoch nicht jedes Mal von vorne durchlaufen werden muß.
- Wir verfolgen die 1. Idee für die Markierung
  - daher boolescher Wert besucht enthält den Wert der für die Markierung genutzt wird

Die Klasse KnotenListe

```
class KnotenListe {
  …
  private KnotenElem position = null;
  Knoten InitIteration() {
      position = kopf;
      return position.GibVerweis();
  }
  Knoten GeheWeiter() {
      position = (position != null?
            position.GibWeiter(): null);
      return (position != null?
            position.GibVerweis(): null);
  }
}
```

Merker für die aktuelle Position in der Adjazenzliste

```
zur Erinnerung: class KnotenElem {
      Knoten verweis; KnotenElem weiter; ...
```

Tiefensuche in gerichteten Graphen

```
class Graph {
  …
  void Durchlaufe() {
      boolean besucht = !erster.GibWarDa();
      Knoten inListe = erster;
      while (inListe != null) {
            System.out.println("Teilgraph:");
            if (inListe.GibWarDa() != besucht)
                  BesucheTeil(inListe, besucht);
            inListe = inListe.GibWeiter();
      }
  }
```

Negation der aktuellen Markierung wird neue Markierung!

Typischer Listendurchlauf mit WHILE Schleife

Tiefensuche in gerichteten Graphen

```
private void BesucheTeil(Knoten aktuell,
                         boolean besucht) {
  KnotenListe adjazenz = aktuell.GibAdjazenz();
  Knoten inAdjazenz    = adjazenz.InitIteration();

  aktuell.SetzeWarDa(besucht);

  System.out.println(aktuell.GibWert());

  while (inAdjazenz != null) {
      if (inAdjazenz.GibWarDa() != besucht)
          BesucheTeil(inAdjazenz, besucht);
      inAdjazenz = adjazenz.GeheWeiter();
  }
}
```

Zwischenstand: Graphen

- Darstellung von Graphen mittels Adjazenzlisten und -matrizen
- Algorithmen basieren
  - auf Graphtraversierungsalgorithmen: DFS, BFS
    - wegen Zyklen Markierung von Knoten erforderlich
    - leistungsfähig: Tarjans Algorithmus für maximale starke Zusammenhangskomponenten arbeitet mit DFS
    - werden durch Listenoperationen (Liste aller Knoten, Adjazenzlisten) unübersichtlich, Beispiel: DFS Enumeration aller Knoten
  - können aber auch auf Matrixoperationen, z.B. Addition/Multiplikation, Potenzieren von Matrizen beruhen
- interessante Fragestellungen im Zusammenhang mit Graphen
  - alle kürzesten Wege: mit Floyd-Warshall Algorithmus in $O(n^3)$
  - ein kürzester Weg: mit Dijkstras Algorithmus in $O(n^2)$
  - Travelling Salesman Problem: mit exponentiellem Aufwand lösbar

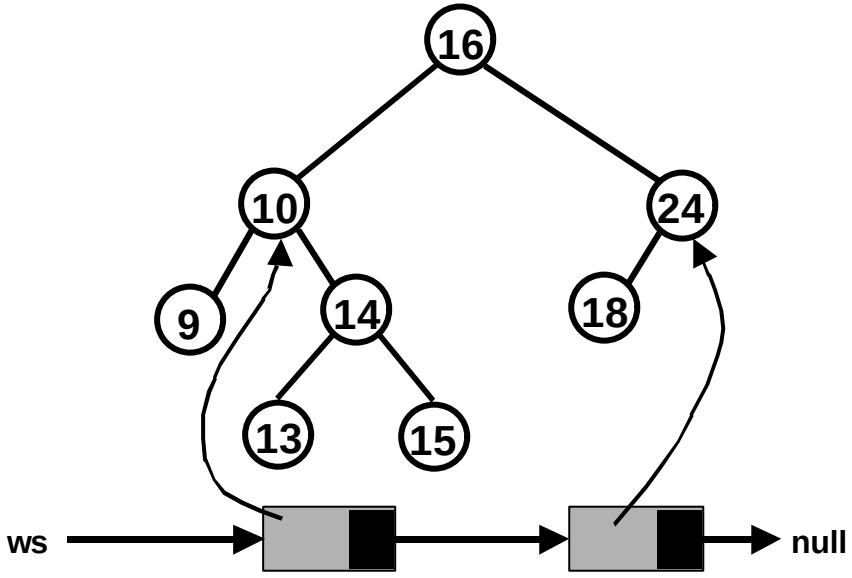
Warteschlangen, Queues

- Queues sind wichtig für die Abarbeitung von Aufgaben in der Reihenfolge FIFO (first-in-first-out), wie an der Supermarktkasse
- Typische Methoden:
  - Prüfen, ob Schlange komplett gefüllt (bei beschränkten Schlangen)
  - Prüfen, ob Schlange leer
  - Anfügen (FIFO: am Ende)
  - Entfernen (am Anfang)
  - Liefern des ersten Elementes
  - Ermittlung der Länge
- Realisierung: mit linearer Liste
  - Zugriff nur mittels der genannten Methoden

    => keine weitern, nicht Queue-kompatiblen Manipulationen möglich, z.B. vordrängeln mangels passender Funktionen ausgeschlossen.

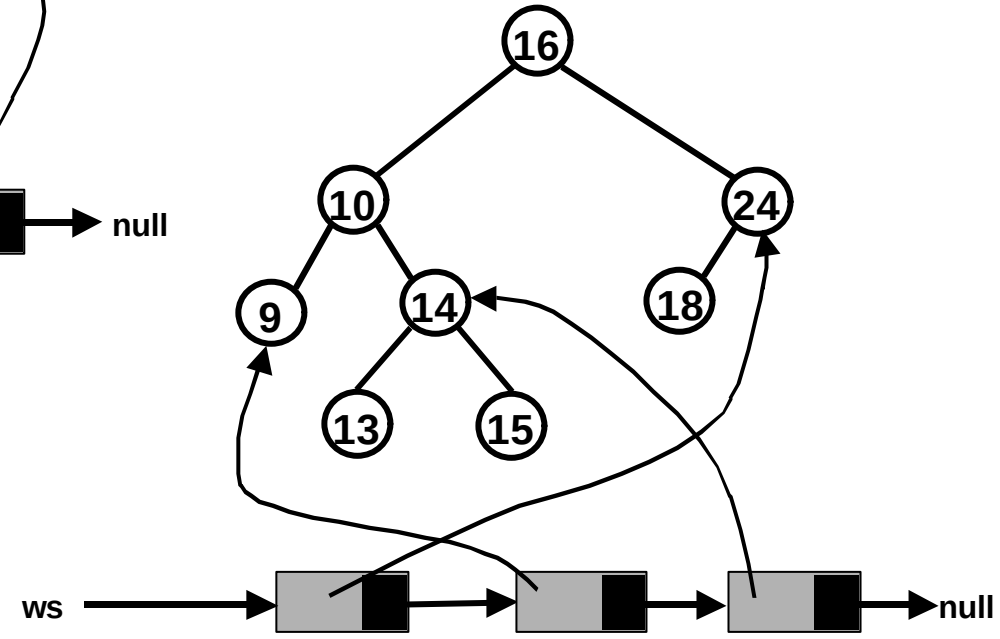
Beispiel: Breitendurchlauf

Bei Bearbeitung eines Knotens werden Nachfolgeknoten im Baum in die Queue eingefügt.



vor Besuch des Knotens 10

nach Besuch des Knotens 10

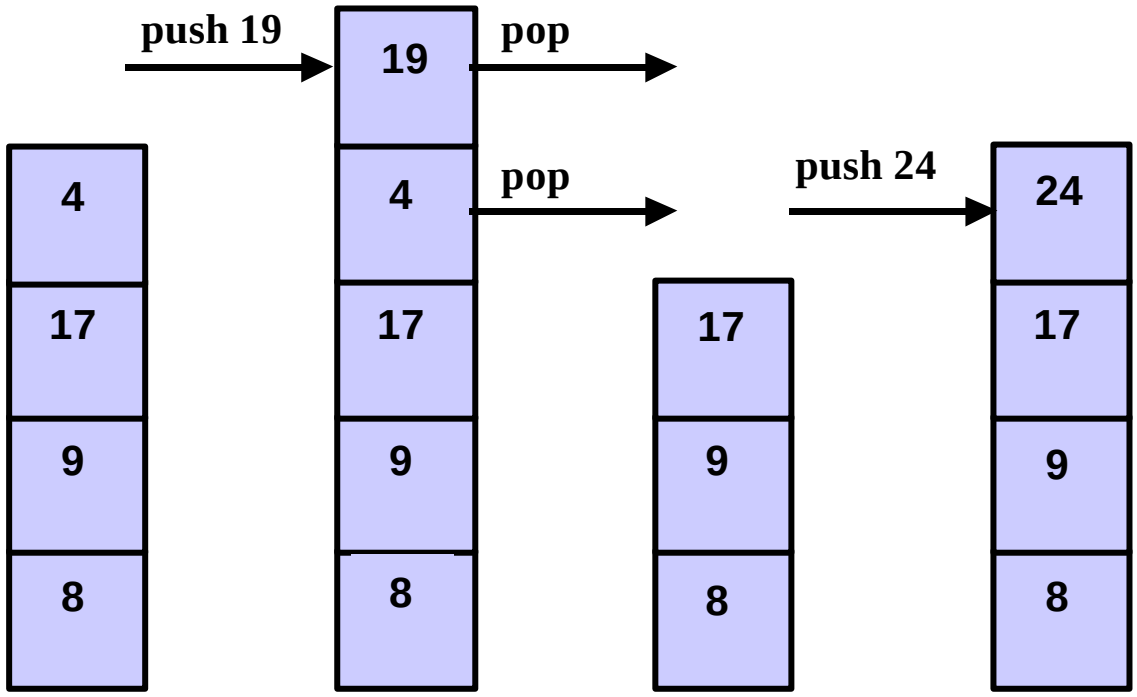


Stapel, Stacks

- Weiteres Beispiel für einen abstrakten Datentyp und den Umgang mit verketteten Listen.
- Elemente dieser Datenstruktur werden nach dem LIFO-Prinzip (Last In First Out) verwaltet.
  - Das letzte in einem Stack abgelegte Element wird als erstes herausgenommen
- Realisierung analog zu Queues mit linearer Liste
  - wesentliche Änderung: Einfügen (LIFO: am Anfang)
- Methoden für Stacks
  - Initialisieren eines leeren Stacks.
  - Testen, ob der Stack leer ist.
  - Ablegen eines Elements auf dem Stack (**push**-Operation)
  - Inspektion des obersten Elementes (**top**-Operation)
  - Entfernen des obersten Elementes des Stacks (**pop**-Operation)

  => einige Methoden aus Queues müssen passend umbenannt werden, Einfügen muss für „push“ zu Einfügen am Anfang abgewandelt werden.

Darstellung der Stack-Methoden



Die Klasse Stack

```
class Stack {
private Element derStack;

Stack() { derStack = null; }
Stack(int i) { derStack = new Element(i); }

boolean empty() { return derStack == null; }

void push(int i) { derStack = new Element(i, derStack); }

int pop() {
   int topWert = (derStack != null ? derStack.GibWert() : 0);
   derStack = derStack.GibWeiter();
   return topWert;
}

int top() {
   return (derStack != null ? derStack.GibWert() : 0);
}
}
```

java.util.Stack

Klasse `java.util.Stack extends java.util.Vector`

- Stapel für Objekte mit allen geerbten Funktionalitäten von Vector
- Sehr mächtig und sehr langsam
  => für reine Stackfunktionalität Eigenimplementierung besser
- `public Stack()`
  - Konstruktor für leeren Stack
- `public Object push(Object item)`
  - Legt Objekt auf Stapelspitze
- `public Object pop() throws EmptyStackException`
  - Holt Objekt von Stapelspitze (und entfernt es)
- `public Object peek() throws EmptyStackException`
  - Gibt Referenz von Objekt auf Stapelspitze, ohne es zu entfernen
- `public boolean empty()`
  - Testet, ob Stack leer ist

java.util.Stack - Beispiel

```
java.util.Stack stapel = new java.util.Stack();
stapel.push("Text");
stapel.push(new Integer(2));
stapel.push(new Double(1e100));

System.out.println("Gesamt: " + stapel);
try {
    while (!stapel.empty())
        System.out.println("Objekt: " + stapel.pop());
} catch (java.util.EmptyStackException e) {
}
```

Ausgabe:

```
Gesamt: [Text, 2, 1.0E100]
Objekt: 1.0E100
Objekt: 2
Objekt: Text
```

Mengen

Darstellung von Mengen:

- natürlich geeignet: Listen, Bäume,

  aber Aufwand für Einfügen, Suchen, Löschen, Aufzählen

(Wie groß ist der Aufwand für eine Suchoperation in Listen, Bäumen ?)

wir betrachten zwei zusätzliche Varianten

- Charakteristische Vektoren:

  bei einer bekannten Obermenge wird durch einen booleschen Vektor festgehalten, welche Elemente in einer konkreten Teilmenge auftreten

- Hashing-Tabellen

  eine Funktion bildet die Beschreibung eines Elementes auf eine Position in einem Array ab, an der dann vermerkt wird, ob das Element in der Menge enthalten ist.

  Folgeproblem: Konfliktbehandlung bei nicht-injektiven Funktionen

Charakteristische Vektoren

- Für jedes potentielle Mengenelement wird in einem Array-Eintrag des Typs Boolean festgehalten, ob es in der Menge ist oder nicht.
- Vorteile:
  - einfache Prüfung auf Enthaltensein
  - einfache Durchführung von Mengenoperationen
- Nachteil:
  - enorme Platzverschwendung, gerade bei vielen potentiellen Elementen und kleinen Mengen
- Beispiel:
  - Grundmenge: alle potentiellen 600 Programmierungsstudenten (Erstsemester Informatik, Physik, Mathe)
  - Menge der am Vorlesungsende regelmäßig teilnehmenden Studierenden, voraussichtlich 300.
  - Studierende seien gegeben durch Name, Fach
  - Wir nehmen an, dass der Name eindeutig ist.
- Das Prüfen, ob Studierende am Ende von Programmierung und RS noch in der Vorlesung sind, ist dann eine einfach Bitoperation (UND)

Hashing

- Die begrenzte Einsetzbarkeit von charakteristischen Vektoren zeigt sich an sehr großen Mengen.
- Beispiel:
  - Grundmenge: Liste aller Wörter im Duden
  - Darzustellende Menge: die Menge aller Wörter auf den Prog-Folien
- Wenn wir nun die Speicherplatzverschwendung begrenzen wollen, dann stellen wir nicht mehr für jedes potentielle Element der Menge einen Speicherplatz zur Verfügung, sondern nur noch eine kleine Anzahl von Speicherplätzen, auf die die Elemente der darzustellenden Menge abgebildet werden.
- Eine Hash-Funktion bildet die darzustellende Menge (=Teilmenge der Grundmenge) auf eine feste Anzahl von Speicherplätzen ab.
- Da man nicht für jedes Element der Grundmenge einen Speicherplatz hat, werden mehrere Elemente der Grundmenge auf die gleiche Position abgebildet.

Hashing

- Sei B die Anzahl der Speicherplätze.
- Eine Hash-Funktion h arbeitet auf den Elementen der Grundmenge und bildet diese auf einen Wert zwischen 0 und B-1 ab. Auf Grund dieser Eigenschaft nennt man Hash-Funktionen auch Schlüsseltransformationen.
- B wird die Anzahl der „Buckets“ der Hash Table genannt.
- h(x) ist der Bucket, in dem das Element x der Grundmenge zu finden ist.
- Beispiele für Hashfunktionen
  - Grundmenge: Integerzahlen, $h(x) = x \bmod B$
  - Grundmenge: Zeichenketten, sei x die Summe der Integer-Werte der einzelnen Zeichen aus String s, dann $h(string) = x \bmod B$

Hashing

- Qualität einer Hash-Funktion:
  - gleichmäßige Verteilung der Elemente der Grundmenge über die Einträge der Tabelle, d.h. für beliebige darzustellende Teilmengen aus der Grundmenge soll eine gleichmäßige Verteilung auf die Buckets stattfinden.
- Konkreter:
  - Eine Hash-Funktion sollte surjektiv sein, also auf alle Buckets abbilden.
  - Sie sollte die Elemente der Grundmenge gleichmäßig auf die Buckets verteilen.
  - Sie sollte effizient zu berechnen sein.
- Lastfaktor einer Hash Table:
  - bei B Speicherplätzen und einer Kardinalität M der darzustellenden Menge ist der Lastfaktor M/B.

Hashing

- Problem:

  Kollision: 2 Elemente haben gleichen Wert der Hashfunktion, hash(x)=hash(y), d.h. die Hashfunktion ist nicht injektiv

- Behandlung:
  - offenes Hashing:

    Einträge der Tabelle sind Verweise auf lineare Listen, in denen die Elemente abgelegt werden
  - geschlossenes Hashing:

    Bucket kann nur eine kleine, fest definierte Anzahl von Elementen aufgenommen werden.
    - bei Überschreiten: rehashing-Verfahren Suchen freien Eintrag in der Tabelle
    - Risiko: rehashing kann kaskadieren, Einfügen und Suchen von Elementen kann dann aufwendig werden.

Zusammenfassung

- Dynamische Datenstrukturen sind wesentlich für fast alle Informatik-Probleme.
- Hier betrachtet:
  - Listen, Bäume,
  - Graphen: Traversieren (Tiefen/Breitensuche), transitive Hülle, kürzeste Wege, Travelling Salesman Problem
  - Queues, Stacks
  - Mengen: charakteristische Vektoren, Hashing
- Dynamische Strukturen verfügen über strukturell ähnliche Methoden und werden häufig in Bibliotheken bereits unterstützt,

  z.B. in Java im Paket java.util
  - Listen werden mit `java.util.Vector` realisiert
  - Stapel werden mit `java.util.Stack` realisiert
  - Realisierung von Tabellen (mit zwei Spalten) mit `java.util.Hashtable`
  - Durchlauf durch Listen mit `java.util.Enumeration`