Keller, Schlangen und Listen

Einfach verkettete Listen

Eine Zeiger-Implementierung von einfach-verketteten Listen, also Listen mit Vorwärtszeigern.

```
//Deklarationsdatei liste.h fuer einfach-verkettete Listen.
enum boolean {False, True };
class liste{

private:
   typedef struct Element {
     int data;
     Element *next;};
   Element *head, *current;

public:    liste( ) // Konstruktor
    { head = new Element; current = head; head->next = 0; }
```

Public: Die weiteren Operationen

- **void insert(int data):**
  Ein neues Element mit Wert data wird nach dem gegenwärtigen Element eingefügt. Der Wert von current ist unverändert.
- **void remove( ):** Das dem gegenwärtigen Element folgende Element wird entfernt. Der Wert von current ist unverändert.
- **void movetofront( ):** current erhält den Wert head.
- **void next( ):** Das nächste Element wird aufgesucht. Dazu wird current um eine Position nach rechts bewegt.
- **boolean end( ):** Ist das Ende der Liste erreicht?
- **boolean empty( ):** Ist die Liste leer?
- **int read( ):** Gib das Feld data des nächsten Elements aus.
- **void write(int wert):** Überschreibe das Feld data des nächsten Elements mit der Zahl wert.
- **void search(int wert):** Suche, rechts von der gegenwärtigen Zelle, nach der ersten Zelle z mit Datenfeld wert. Der Zeiger current wird auf den Vorgänger von z zeigen.

Eigenschaften

- Jede Liste besitzt stets eine leere Kopfzelle: **liste()** erzeugt diese Zelle, nachfolgende Einfügungen können nur nach der Kopfzelle durchgeführt werden.
- Alle Operationen –bis auf search– werden in konstanter Zeit $O(1)$ unterstützt. Die search-Funktion benötigt möglicherweiser Zeit proportional zur Länge der Liste.

The good and the bad

+ Die Größe der Liste ist proportional zur Anzahl der gespeicherten Elemente: Die Liste passt sich der darzustellenden Menge an.
- Die Suche dauert viel zu lange.

Die Addition dünn besetzter Matrizen

A und B sind Matrizen mit n Zeilen und m Spalten.
Berechne die Summe $C = A + B$.

- Das Programm

```
for (i=0 ; i < n; i++)
  for (j=0 ; j < m; j++)
    C[i,j] = A[i,j] + B[i,j];
```

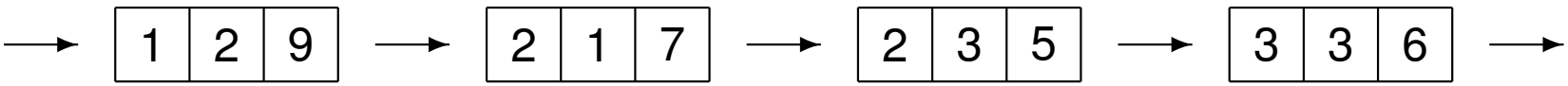
  bestimmt C in Zeit $O(n \cdot m)$.

- **Ziel**: Bestimme C in Zeit $O(a + b)$, wobei a und b die Anzahl der von Null verschiedenen Einträge von A und B ist.

Listendarstellung von Matrizen

- Stelle A und B durch einfach verkettete Listen L_A und L_B in Zeilenordnung dar.
- Jedes Listenelement speichert neben dem Wert eines Eintrags auch die Zeile und die Spalte des Eintrags.

Die Zeilenordnung für $A \equiv \begin{bmatrix} 0 & 9 & 0 \\ 7 & 0 & 5 \\ 0 & 0 & 6 \end{bmatrix}$ ist



Der Algorithmus

(1) Beginne jeweils am Anfang der Listen L_A und L_B.

(2) Solange beide Listen nicht-leer sind, wiederhole

  (a) das gegenwärtige Listenelement von L_A (bzw. L_B) habe die Koordinaten (i_A, j_A) (bzw. (i_B, j_B)).

  (b) Wenn $i_A < i_B$ (bzw. $i_A > i_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

  (c) Wenn $i_A = i_B$ und $j_A < j_B$ (bzw. $j_A > j_B$), dann füge das gegenwärtige Listenelement von L_A (bzw. L_B) in die Liste L_C ein und gehe zum nächsten Listenelement von L_A (bzw. L_B).

  (d) Wenn $i_A = i_B$ und $j_A = j_B$, dann addiere die beiden Einträge und füge die Summe in die Liste L_C ein. Die Zeiger in beiden Listen werden nach rechts bewegt.

(3) Wenn die Liste L_A (bzw. L_B) leer ist, kann der Rest der Liste L_B (bzw. L_A) an die Liste L_C angehängt werden.

Deques, Stacks und Queues

- Der abstrakte Datentyp **Stack** unterstützt die Operationen
  - `pop` : Entferne den jüngsten Schlüssel.
  - `push` : Füge einen Schlüssel ein.

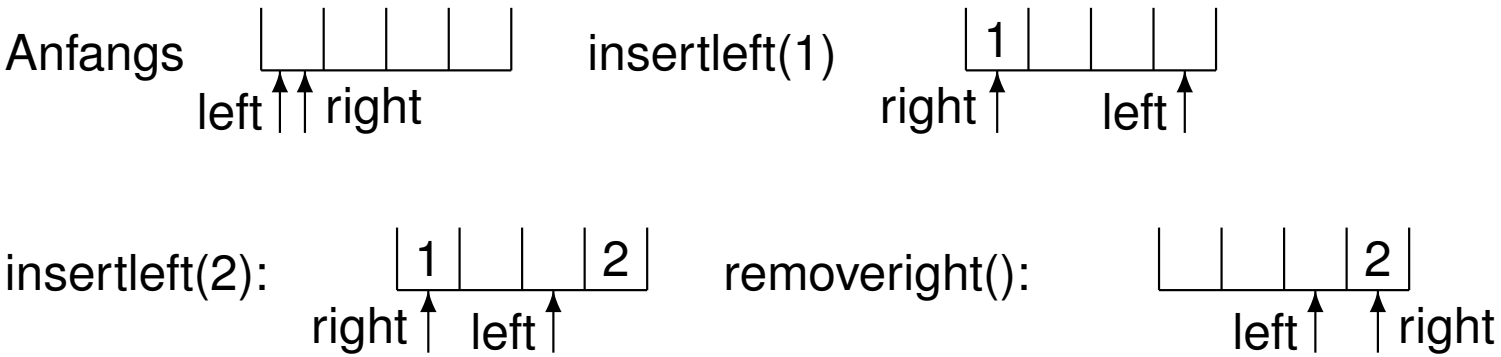
  Ein wichtiges Anwendungsgebiet ist die nicht-rekursive Implementierung rekursiver Programme.
- Eine **Queue** modelliert das Konzept einer Warteschlange und unterstützt die Operationen
  - `dequeue`: Entferne den ältesten Schlüssel.
  - `enqueue` : Füge einen Schlüssel ein.
- Stacks und Queues werden von einem **Deque** verallgemeinert. Die folgenden Operationen werden unterstützt:
  - `insertleft` : Füge einen Schlüssel am linken Ende ein.
  - `insertright` : Füge einen Schlüssel am rechten Ende ein.
  - `removeleft` : Entferne den Schlüssel am linken Ende.
  - `removeright` : Entferne den Schlüssel am rechten Ende.

Wie werden Deques und Listen implementiert?

Die Implementierung eines Deque

- Benutze ein an beiden Enden „zusammengeschweißtes“ 1-dimensionales Array.
  - Der Inhalt des Deques wird jetzt entsprechend den Operationen „über den entstandenen Ring“ geschoben.
- Benutze zwei Positionsvariable **left** und **right**: **left** steht jeweils **vor** dem linken Ende und **right** jeweils **auf** dem rechten Ende.
- **Forderung**:

  Die Zelle mit Index **left** ist stets leer und anfänglich ist **left = right**.



Eine Implementierung von Listen durch Arrays (1/2)

Eine einfach verkettete Liste bestehe aus Zellen mit einem **integer-Datenfeld** und einem **Vorwärtszeiger**.
Zu keinem Zeitpunkt möge die Liste mehr als n Elemente besitzen.

- **Daten** und **Zeiger** seien Arrays der Größe n.
- Wenn ein Listenelement den „Index i erhält“, dann ist **Daten**$[i]$ der Wert des Listenelements und **Zeiger**$[i]$ der Index des rechten Nachbarn.
- Für die Speicherverwaltung benutze eine zusätzliche Datenstruktur **Frei**, die die Menge der freien Indizes verwaltet.

  Zu Anfang ist **Frei**$= \{0, \ldots, n-1\}$.

Eine Implementierung von Listen durch Arrays (2/2)

- Wenn ein Element mit Wert w nach einem Element mit Index i einzufügen ist:

  Wenn **Frei** nicht-leer ist, dann entnimm einen freien Index j und setze **Daten**$[j] = w$, **Zeiger**$[j] =$ **Zeiger**$[i]$ und **Zeiger**$[i] = j$.

- Wenn ein Element zu entfernen ist, dessen Vorgänger den Index i besitzt:

  Füge $j =$ **Zeiger**$[i]$ in die Datenstruktur Frei ein und setze **Zeiger**$[i] =$ **Zeiger**$[j]$.

- Welche Datenstruktur ist für **Frei** zu wählen?

  Jede Datenstruktur, die es erlaubt, Elemente einzufügen und zu entfernen, tut's: ein Stack, eine Queue oder ein Deque ist OK.

Bäume

Bäume

Gewurzelte Bäume für die hierarchische Strukturierung von Daten.

- Gewurzelte Bäume als konzeptionelle Hilfsmittel für
  - von einem Benutzer angelegte Verzeichnisse,
  - als Nachfahrenbaum
  - oder für die Veranschaulichung rekursiver Programme (Rekursionsbaum oder Baum einer Rekursionsgleichung).

- Gewurzelte Bäume als Datenstrukturen von Algorithmen:
  - in der Auswertung arithmetischer Ausdrücke,
  - in der Syntaxerkennung mit Hilfe von Syntaxbäumen,
  - im systematischen Aufzählen von Lösungen (Entscheidungsbäume)
  - oder in der Lösung von Suchproblemen.

Was ist ein gewurzelter Baum?

Ein gewurzelter Baum T wird durch eine **Knotenmenge** V und eine **Kantenmenge** $E \subseteq V \times V - \{(i,i) \mid i \in V\}$ dargestellt.
Die gerichtete Kante (i,j) führt **von** i **nach** j.

Wann ist T ein gewurzelter Baum?

- T muss genau einen Knoten r besitzen, in den keine Kante hineinführt. r heißt die **Wurzel** von T.
- In jeden Knoten darf höchstens eine Kante hineinführen
- und jeder Knoten muss von der Wurzel aus erreichbar sein.

(Ab jetzt sprechen wir nur von Bäumen und meinen gewurzelte Bäume.)

Eine (knotendisjunkte) Vereinigung von Bäumen heißt ein **Wald**.

Zentrale Begriffe

- Wenn (v, w) eine Kante ist, dann nennen wir v den **Elternknoten** von w und sagen, dass w ein **Kind** von v ist.
  - Die anderen Kinder von v heißen **Geschwister** von w.
- **Aus-Grad** (v) ist die Anzahl der Kinder von v.
  - Einen Knoten b mit Aus-Grad $(b) = 0$ bezeichnen wir als **Blatt**.
  - T heißt k-**när**, falls der Aus-Grad aller Knoten höchstens k ist. Für $k = 2$ sprechen wir von **binären Bäumen**.
- Ein **Weg** von v nach w ist eine Folge $(v_0, \ldots, v_m)$ von Knoten mit $v_0 = v$, $v_m = w$ und $(v_i, v_{i+1}) \in E$ für alle i $(0 \leq i < m)$.
  - v ist ein **Vorfahre** von w und w ein **Nachfahre** von v.
  - Die **Länge** des Weges ist m, die Anzahl der Kanten des Weges.
  - **Tiefe**(v) ist die Länge des (eindeutigen) Weges von r nach v.
  - **Höhe**(v) ist die Länge des längsten Weges von v zu einem Blatt.
- T heißt **geordnet**, falls für jeden Knoten eine Reihenfolge der Kinder vorliegt.

Operationen auf Bäumen

(1) `Wurzel`: Bestimme die Wurzel von T.

(2) `Eltern`(v): Bestimme den Elternknoten des Knotens v in T.
Wenn $v = r$, dann ist der Null-Zeiger auszugeben.

(3) `Kinder`(v): Bestimme die Kinder von v. Wenn v ein Blatt ist, dann ist der Null-Zeiger als Antwort zu geben.

(4) Für binäre geordnete Bäume:

  (4a) `LKind`(v): Bestimme das linke Kind von v.
  (4b) `RKind`(v): Bestimme das rechte Kind von v.
  (4c) Sollte das entsprechende Kind nicht existieren, ist der Null-Zeiger als Antwort zu geben.

(5) `Tiefe`(v): Bestimme die Tiefe von v.

(6) `Höhe`(v): Bestimme die Höhe von v.

(7) `Baum`($v, T_1, \ldots, T_m$): Erzeuge einen geordneten Baum mit Wurzel v und Teilbäumen $T_1, \ldots, T_m$.

(8) `Suche`(x): Bestimme alle Knoten mit Wert x.

Das Eltern-Array
(wenn nur Vorfahren interessieren)

Das Eltern-Array

Annahme: Jeder Knoten besitzt eine Zahl aus $\{1, \ldots, n\}$ als Namen und zu jedem $i \in \{1, \ldots, n\}$ gibt es genau einen Knoten mit Namen i.

- Ein integer-Array **Baum** speichert für jeden Knoten v den Namen des Elternknotens von v:

  $$\text{Baum}[v] = \text{Name des Elternknotens von } v.$$

  Wenn $v = r$, dann setze $\text{Baum}[v] = 0$.

- Das Positive:
  - \+ schnelle Bestimmung des Elternknotens (Zeit $= O(1)$)
  - \+ und schnelle Bestimmung der Tiefe von v (Zeit $= O(\text{Tiefe}(v))$).
  - \+ Minimaler Speicherplatzverbrauch:
    Bäume mit n Knoten benötigen Speicherplatz n.
- Das Negative: für die Bestimmung der Kinder muss der gesamte Baum durchsucht werden (Zeit = O(Anzahl Knoten).)

Die Binärbaum-Implementierung (die ideale Datenstruktur für Binärbäume)

Die Binärbaum-Implementierung

Ein Knoten wird durch die Struktur

```
struct Knoten {
  int wert;
  Knoten *links, *rechts; };
```

dargestellt.

- Wenn der Zeiger z auf die Struktur des Knotens v zeigt,
  - dann ist $z \to \mathit{wert}$ der *Wert* von v und
  - $z \to \mathit{links}$ (bzw. $z \to \mathit{rechts}$) zeigt auf die Struktur des linken (bzw. rechten) Kindes von v.
  - Der Zeiger wurzel zeigt auf die Struktur der Wurzel des Baums.
- Speicherbedarf:
  - $2n$ Zeiger (zwei Zeiger pro Knoten) und
  - n Zellen (eine Zelle pro Knoten).

Die Binärbaum-Implementierung: Stärken und Schwächen

+ Die **Kinderbestimmung** gelingt schnellstmöglich, in Zeit $O(1)$.
+ **Höhe** (v, T) wird angemessen unterstützt mit der Laufzeit

$$O(\text{Anzahl der Knoten im Teilbaum mit Wurzel } v).$$

Begründung später.

- Für die **Bestimmung des Elternknotens** muss möglicherweise der gesamte Baum durchsucht werden!
- Die **Bestimmung der Tiefe** ist auch schwierig, da der Elternknoten nicht bekannt ist.

Die Kind-Geschwister Implementierung (die Allzweckwaffe)

Die Kind-Geschwister Implementierung

Ein Knoten wird durch die Struktur

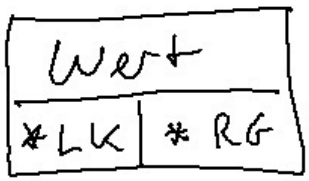
```
typedef struct Knoten {
  int wert;
  Knoten *LKind, *RGeschwister; };
```

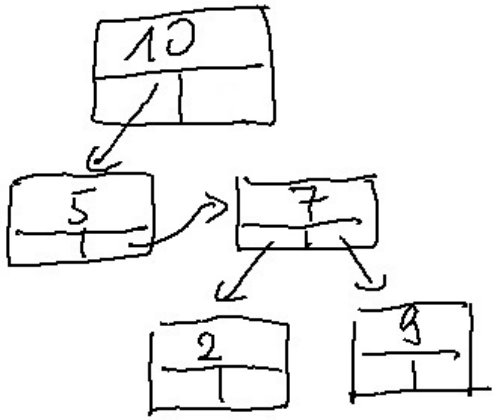
dargestellt.

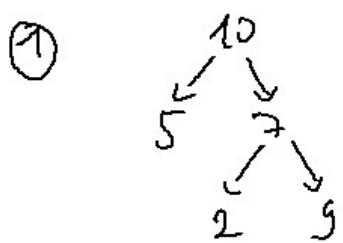
- Wenn der Zeiger z auf die Struktur des Knotens v zeigt,
  - dann ist $z \to \mathit{wert}$ der *Wert* von v und
  - $z \to$ LKind (bzw. $z \to$ RGeschwister) zeigt auf die Struktur des linken Kindes, bzw. des rechten Geschwisterknotens von v.
  - Der Zeiger wurzel zeigt wieder auf die Struktur der Wurzel des Baums.
- Im Vergleich zur Binärbaum-Darstellung:
  - Ähnliches Laufzeitverhalten und ähnliche Speichereffizienz,
  - aber jetzt lassen sich alle Bäume und nicht nur Binärbäume darstellen!

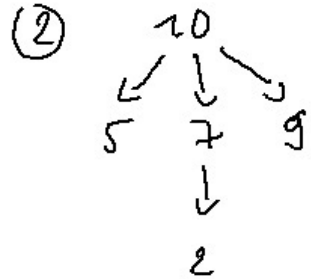
Kind-Geschwister demogr.

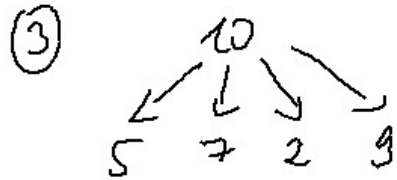
Welchem Baum entspricht die Datenstruktur?











Auflösung: (2)

Postorder, Präoder und Inorder

Suche in Bäumen: Postorder, Präorder und Inorder

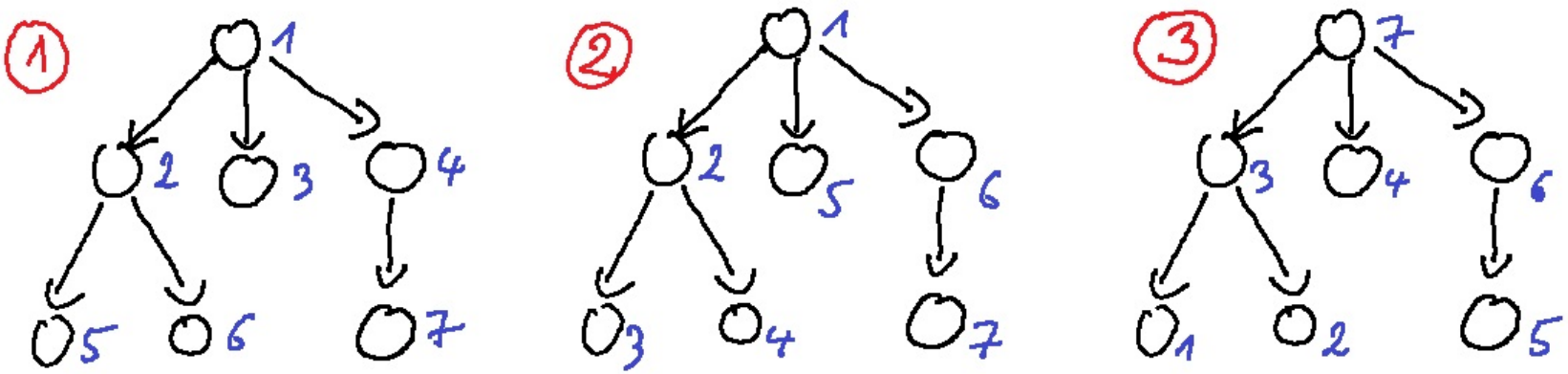
Sei T ein geordneter Baum mit Wurzel r und Teilbäumen $T_1, \ldots, T_m$.

- **Postorder**: Durchlaufe rekursiv die Teilbäume $T_1, \ldots, T_m$ nacheinander. Danach wird die Wurzel r besucht.
- **Präorder**: besuche zuerst r und danach durchlaufe rekursiv die Teilbäume $T_1, \ldots, T_m$.
- **Inorder**: Durchlaufe zuerst T_1 rekursiv, sodann die Wurzel r und letztlich die Teilbäume $T_2, \ldots, T_m$ rekursiv.

```
void praeorder (Knoten *p)
{
   if (p != nullptr)
   { cout « p→ wert;
     for ( Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
       praeorder(q);
   }
}
```

Traversierungen demogr.

Welches ist eine Präorder-Traversierung?



Auflösung: (2)

Bestimmung der Tiefe und Höhe von Knoten

Die Struktur eines Knoten besteht aus den Feldern

tiefe, hoehe, wert, LKind und RGeschwister.

```
tiefe(wurzel,-1);
void tiefe (Knoten *p, int t)
{ t = t+1;
   if (p != nullptr)
   { p→tiefe = t;
     for (Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
         tiefe(q,t);}}

void hoehe (Knoten *p)
{ int h=-1;
   if (p != nullptr)
   { for (Knoten *q=p→LKind; q != nullptr; q=q→RGeschwister)
     { hoehe (q); h = max ( h, q → hoehe); }
     p→hoehe = h+1; } }
```

Eine nicht-rekursive Präorder-Implementierung

Der Teilbaum mit Wurzel v ist in Präorder-Reihenfolge zu durchlaufen.

(1) Wir fügen einen Zeiger auf die Struktur von v in einen anfänglich leeren Stack ein.

(2) Solange der Stack nicht-leer ist, wiederhole:

  (a) Entferne das zuoberst liegende Stack-Element w mit Hilfe der Pop-Operation.
  /$*$ w wird besucht. $*$ /

  (b) Die Kinder von w werden in umgekehrter Reihenfolge in den Stack eingefügt.
  /$*$ Durch die Umkehrung der Reihenfolge werden die Bäume später in ihrer natürlichen Reihenfolge abgearbeitet. $*$ /

Die Laufzeit ist linear in der Knotenzahl n. Warum?

- Jeder Knoten wird genau einmal in den Stack eingefügt.
- Insgesamt werden also höchstens $O(n)$ Stackoperationen durchgeführt. Stackoperationen dominieren aber die Laufzeit.

Welcher Knoten wird direkt nach v besucht?

- **Postorder**:
  - Das linkeste Blatt im Baum des rechten Geschwisterknotens.
  - Wenn v keinen rechten Geschwisterknoten besitzt, dann wird der Elternknoten von v als nächster besucht.
- **Präorder**:
  - Das linkeste Kind von v, wenn v kein Blatt ist.
  - Wenn v ein Blatt ist, dann das erste nicht-besuchte Kind des tiefsten, nicht vollständig durchsuchten Vorfahren von v.

Graphen

Graphen

(a) Ein **ungerichteter Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge

$$E \subseteq \{\{i, j\} \mid i, j \in V, i \neq j\}$$

- Die **Endpunkte** u, v einer ungerichteten Kante $\{u, v\}$ sind gleichberechtigt.
- u und v heißen **Nachbarn**.

(b) Für die Kantenmenge E eines **gerichteten Graphen** $G = (V, E)$ gilt

$$E \subseteq \{(i, j) \mid i, j \in V, i \neq j\}$$

- Der Knoten u ist **Anfangspunkt** und der Knoten v **Endpunkt** der Kante (u, v).
- v heißt auch ein **direkter Nachfolger** von u und u ein **direkter Vorgänger** von v.

Warum Graphen?

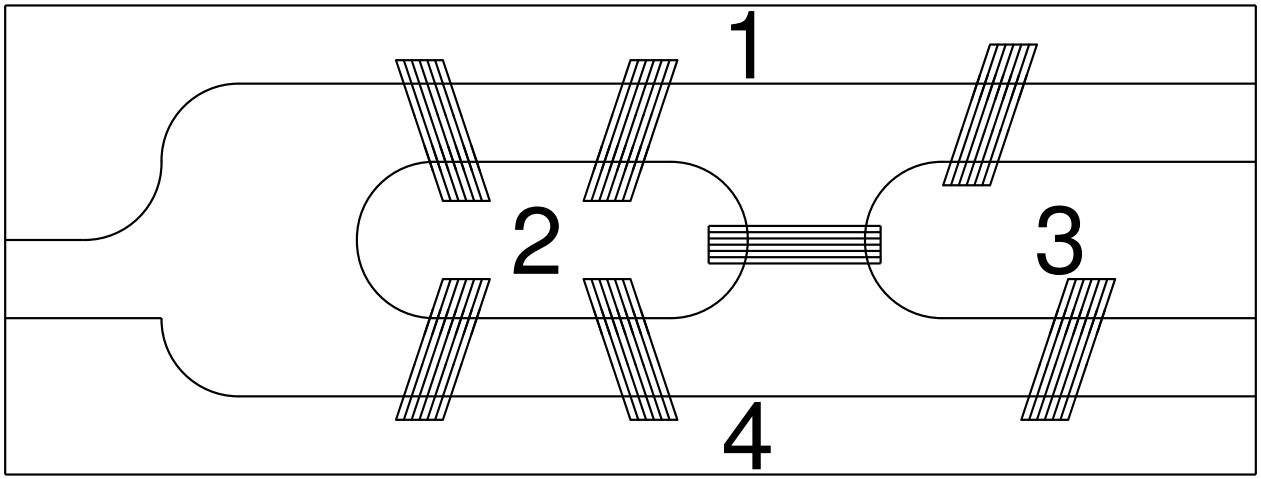
Graphen modellieren

- das **World Wide Web**: Die Knoten entsprechen Webseiten, die (gerichteten) Kanten entsprechen Hyperlinks.
- **Rechnernetzwerke**: Die Knoten entsprechen Rechnern, die (gerichteten und/oder ungerichteten) Kanten entsprechen Direktverbindungen zwischen Rechnern.
- Das **Schienennetz der Deutschen Bahn**: Die Knoten entsprechen Bahnhöfen, die (ungerichteten) Kanten entsprechen Direktverbindungen zwischen Bahnhöfen.

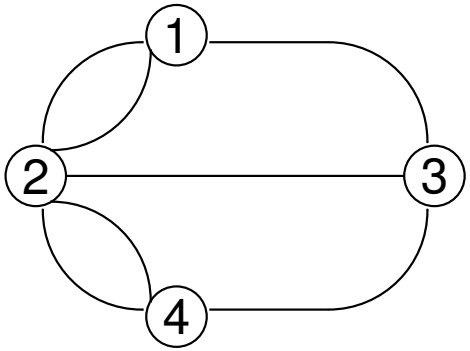
  Bei der Erstellung von Reiseplänen bestimme kürzeste (gewichtete) Wege zwischen einem Start- und einem Zielbahnhof.
- **Schaltungen**: die Knoten entsprechen Gattern, die (gerichteten) Kanten entsprechen Leiterbahnen zwischen Gattern.

Das Königsberger Brückenproblem

Gibt es einen Rundweg durch Königsberg, der alle Brücken über die Pregel genau einmal überquert?



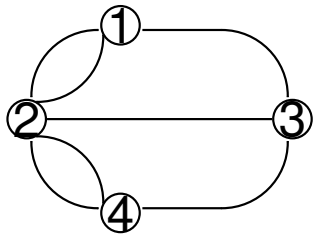
Der zugehörige ungerichtete Graph:



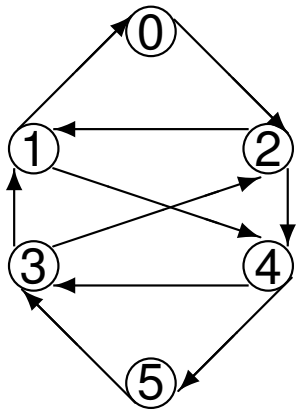
Euler-Kreise

Ein **Euler-Kreis** beginnt in einem Knoten v, durchläuft alle Kanten genau einmal und kehrt dann zu v zurück.

Das Königsberger Brückenproblem besitzt keine Lösung, denn der Graph



hat keinen Euler-Kreis: Ansonsten hätte jeder Knoten eine gerade Anzahl von Nachbarn! Und der folgende Graph?



Wichtige Begriffe

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph.

- Eine Folge $(v_0, v_1, ..., v_m)$ heißt ein **Weg** in G, falls für jedes i $(0 \leq i < m)$
  - $(v_i, v_{i+1}) \in E$ (für gerichtete Graphen) oder
  - $\{v_i, v_{i+1}\} \in E$ (für ungerichtete Graphen).

  Die **Weglänge** ist m, die Anzahl der Kanten. Ein Weg heißt **einfach**, wenn kein Knoten zweimal auftritt.
- Ein Weg heißt ein **Kreis**, wenn $v_0 = v_m$ und $(v_0, ...v_{m-1})$ ein einfacher Weg ist. G heißt **azyklisch**, wenn G keine Kreise hat.
- Ein ungerichteter Graph heißt **zusammenhängend**, wenn je zwei Knoten durch einen Weg miteinander verbunden sind.

Topologisches Sortieren

n Aufgaben $a_0, \ldots, a_{n-1}$ sind auszuführen. Allerdings gibt es eine Menge P von p Prioritäten zwischen den einzelnen Aufgaben.

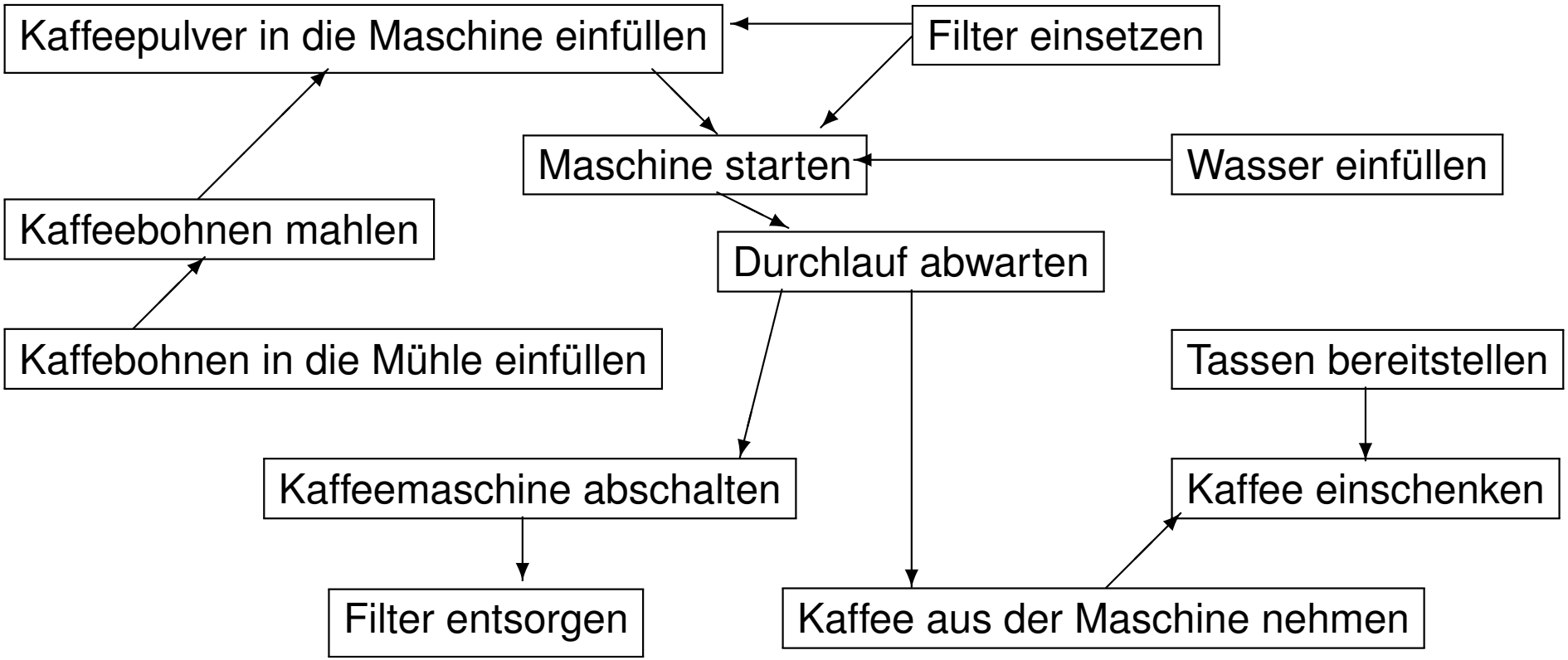
> Die Priorität (i, j) impliziert, dass Aufgabe a_i vor Ausführung der Aufgabe a_j ausgeführt werden muss.

Bestimme eine Reihenfolge, in der alle Aufgaben ausgeführt werden können, bzw. stelle fest, dass eine solche Reihenfolge nicht existiert.

- Eine **graph-theoretische Formulierung** mit Knotenmenge $V = \{0, \ldots, n-1\}$:
  - Knoten i entspricht der Aufgabe a_i.
  - Wir setzen für jede Priorität (i, j) die Kante (i, j) ein.
- Wie ist das **Ziel** zu formulieren?

  Bestimme eine **Reihenfolge $v_1, \ldots, v_i, \ldots, v_n$ der Knoten**, so dass es keine Kante (v_i, v_j) mit $j < i$ gibt.

Kaffeekochen



Die Idee

Eine Aufgabe a_j kann als **erste** Aufgabe ausgeführt werden, wenn es keine Priorität der Form (i, j) in P gibt.

- Ein Knoten v von G heißt eine **Quelle**, wenn In-Grad$(v) = 0$, wenn v also kein Endpunkt einer Kante ist.
- Also bestimme eine Quelle v, führe v aus und entferne v.
- Wiederhole dieses Verfahren, solange G noch Knoten besitzt:

  bestimme eine Quelle v, führe v aus und entferne v.

Welche Datenstrukturen sollten wir verwenden?

Ein erster Versuch

Wir verketten alle p Kanten in einer Liste „Priorität“ und benutzen ein integer Array „Reihenfolge“ sowie zwei boolesche Arrays „Erster“ und „Fertig“ mit jeweils n Zellen.

Zaehler$=0$. Für alle i setze Fertig$[i] =$ falsch.

Wiederhole n-mal:

(0) Setze Erster$[i] =$ wahr genau dann, wenn Fertig$[i] =$ falsch.

(1) Durchlaufe die Liste Priorität. Wenn Kante (i, j) angetroffen wird, setze Erster$[j]$ = falsch.

(2) Bestimme das kleinste j mit Erster$[j]$ = wahr. Danach setze

  (a) Fertig$[j] =$ wahr,
  (b) Reihenfolge[Zaehler++] $= j$ (Aufgabe j wird ausgeführt)
  (c) und durchlaufe die Prioritätsliste: entferne jede Kante (j, k), da a_j eine Ausführung von Aufgabe a_k nicht mehr behindert.

Laufzeit Topologisches Sortieren (1) demogr.

Worst-Case Laufzeit für den ersten Versuch?

- (1) $\Theta(n + p)$
- (2) $\Theta(n \cdot p)$
- (3) $\Theta(n \cdot \log n)$
- (4) $\Theta(n \cdot (n + p))$

Auflösung: (4) $\Theta(n \cdot (n + p))$

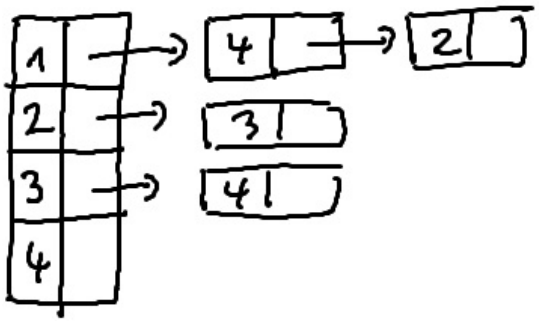
Eine Laufzeitanalyse

- Was ist besonders teuer?
  - In jeder Iteration muss die Liste Priorität vollständig durchlaufen werden: Zeit $= O(p)$.
  - Weiterhin muss das Array Erster jeweils initialisiert werden: Zeit $= O(n)$.
  - Die Laufzeit pro Iteration ist dann durch $O(n + p)$ beschränkt. Die Gesamtlaufzeit ist $O(n \cdot (n + p))$, da wir n Iterationen haben.
- Was können wir verbessern?
  - Wir müssen nur die Kanten entfernen, die im gerade ausgeführten Knoten j beginnen.
  - Warum kompliziert nach der ersten ausführbaren Aufgabe suchen?

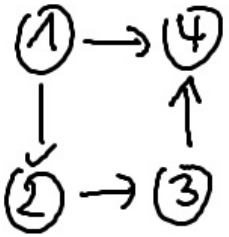
    Eine vorher nicht in Betracht kommende Aufgabe k wird nur interessant, wenn (j, k) eine Priorität ist.

Adjazenzlistendarstellung demogr.

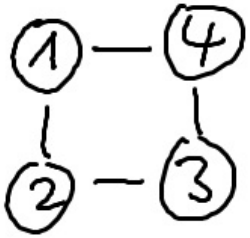
Welcher Graph wird repräsentiert?



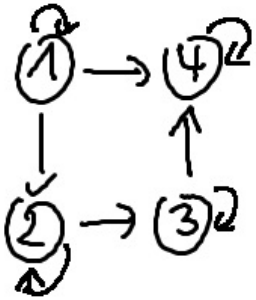
1)



2)



3)



Auflösung: (1)

Der zweite Versuch

Stelle die Prioritäten durch eine Adjazenzliste mit dem Kopf-Array **Priorität** dar.
Benutze ein Array **In-Grad** mit In-Grad[v] $= k$, falls v Endpunkt von k Kanten ist.

(1) Initialisiere die Adjazenzliste Priorität durch Einlesen aller Prioritäten. (Zeit = $O(n+p)$).

(2) Initialisiere das Array In-Grad. (Zeit = $O(n+p)$).

(3) Alle Knoten v mit In-Grad[v] $= 0$ werden in eine **Schlange** eingefügt. (Zeit = $O(n)$).

(4) Setze Zähler $= 0$; Wiederhole solange, bis *Schlange* leer ist:

  (a) Entferne einen Knoten i aus *Schlange*.
  (b) Setze **Reihenfolge** [Zähler++] $= i$.
  (c) Durchlaufe die Liste Priorität [i] und reduziere In-Grad für jeden Nachfolger j von i um 1. Wenn jetzt In-Grad[j] $= 0$, dann füge j in *Schlange*: Aufgabe a_j ist jetzt ausführbar.

Laufzeit Topologisches Sortieren (2) demogr.

Worst-Case Laufzeit für den zweiten Versuch?

- (1) $\Theta(n + p)$
- (2) $\Theta(n \cdot p)$
- (3) $\Theta(n \cdot \log n)$
- (4) $\Theta(n \cdot (n + p))$

Auflösung: (1) $\Theta(n + p)$

Die Analyse

- Die Vorbereitungsschritte (1), (2) und (3) laufen in $O(n+p)$ Schritten ab.
- Ein Knoten wird nur einmal in die Schlange eingefügt. Also beschäftigen sich höchstens $O(n)$ Schritte mit der Schlange.
- Eine Kante (i,k) wird, mit Ausnahme der Vorbereitungsschritte, nur dann inspiziert, wenn i aus der Schlange entfernt wird.
  - Jede Kante wird nur einmal „angefasst“
  - und höchstens $O(p)$ Schritte behandeln Kanten.

Das Problem des topologischen Sortierens wird für einen Graphen mit n Knoten und p Kanten in Zeit $O(n+p)$ gelöst.

Schneller geht's nimmer.

Adjazenzlisten und die Adjazenzmatrix

Graph-Implementierungen

- Welche Datenstruktur sollten wir für die Darstellung eines Graphen G wählen?
- Welche Operationen sollen schnell ausführbar sein?
  - Ist e eine Kante von G?
    Die **Adjazenzmatrix** wird sich als eine gute Wahl herausstellen.
  - Bestimme die Nachbarn, bzw. Vorgänger und Nachfolger eines Knotens.
    Die **Adjazenzlistendarstellung** ist unschlagbar.

Besonders die Nachbar- und Nachfolgerbestimmung ist wichtig, um Graphen zu durchsuchen.

Die Adjazenzmatrix

Für einen Graphen $G = (V, E)$ (mit $V = \{0, ..., n-1\}$) ist

$$A_G[u, v] = \begin{cases} 1 & \text{wenn } \{u, v\} \in E \text{ (bzw. wenn } (u, v) \in E), \\ 0 & \text{sonst} \end{cases}$$

die Adjazenzmatrix A_G von G.

+ Eine Kantenfrage „**ist (u, v) eine Kante?**“ wird sehr schnell beantwortet, nämlich in Zeit $O(1)$.
- Die Bestimmung aller Nachbarn oder Nachfolger eines Knotens v ist hingegen langwierig:
  - Die Zeile von v muss durchlaufen werden.
  - Zeit $\Theta(n)$ ist selbst dann notwendig, wenn v nur wenige Nachbarn hat.
- Speicherplatzbedarf $\Theta(n^2)$ auch für Graphen mit relativ wenigen Kanten:

  Die Datenstruktur passt sich nicht der Größe des Graphen an!

Die Adjazenzliste

G wird durch ein Array A von Listen dargestellt. Die Liste $A[v]$ führt alle Nachbarn von v auf, bzw. alle Nachfolger von v für gerichtete Graphen.

+ Die **Nachbar- bzw. Nachfolgerbestimmung** für Knoten v gelingt in Zeit proportional zur Anzahl der Nachbarn oder Nachfolger.
+ Der benötigte Speicherplatz ist $O(n + |E|)$: Die Datenstruktur passt sich der Größe des Graphen an.
- Für die Beantwortung der Kantenfrage „ist (u, v) eine Kante?“ muss die Liste $A[v]$ durchlaufen werden: Die benötigte Zeit ist also proportional zur Anzahl der Nachbarn oder Nachfolger.

Da die Nachbar- bzw. Nachfolgerbestimmung für das Durchlaufen von Wegen benötigt wird, ist die sich der Größe des Graphen anpassende Adjazenzliste die Datenstruktur der Wahl.

Suche in Graphen: Tiefensuche

Suche in Graphen

Wie durchsucht man ein Labyrinth? Können wir Präorder benutzen?

- Präorder terminiert nur für Wälder.
  - Präorder wird von Kreisen in eine Endlosschleife gezwungen,
  - es erkennt nicht, dass Knoten bereits besucht wurden!
- Können wir Präorder reparieren?

Wie findet man Wege aus einem Labyrinth?

Ein Auszug aus *Umbert Eco's „Der Name der Rose"*.

William von Baskerville und sein Schüler Adson van Melk sind heimlich in die als Labyrinth gebaute Bibliothek eines hochmittelalterlichen Klosters irgendwo im heutigen Norditalien eingedrungen.

Fasziniert von den geistigen Schätzen, die sie beherbergt, haben sie sich nicht die Mühe gemacht, sich ihren Weg zu merken.

Erst zu spät erkennen sie, dass die Räume unregelmäßig und scheinbar wirr miteinander verbunden sind.

Man sitzt fest.

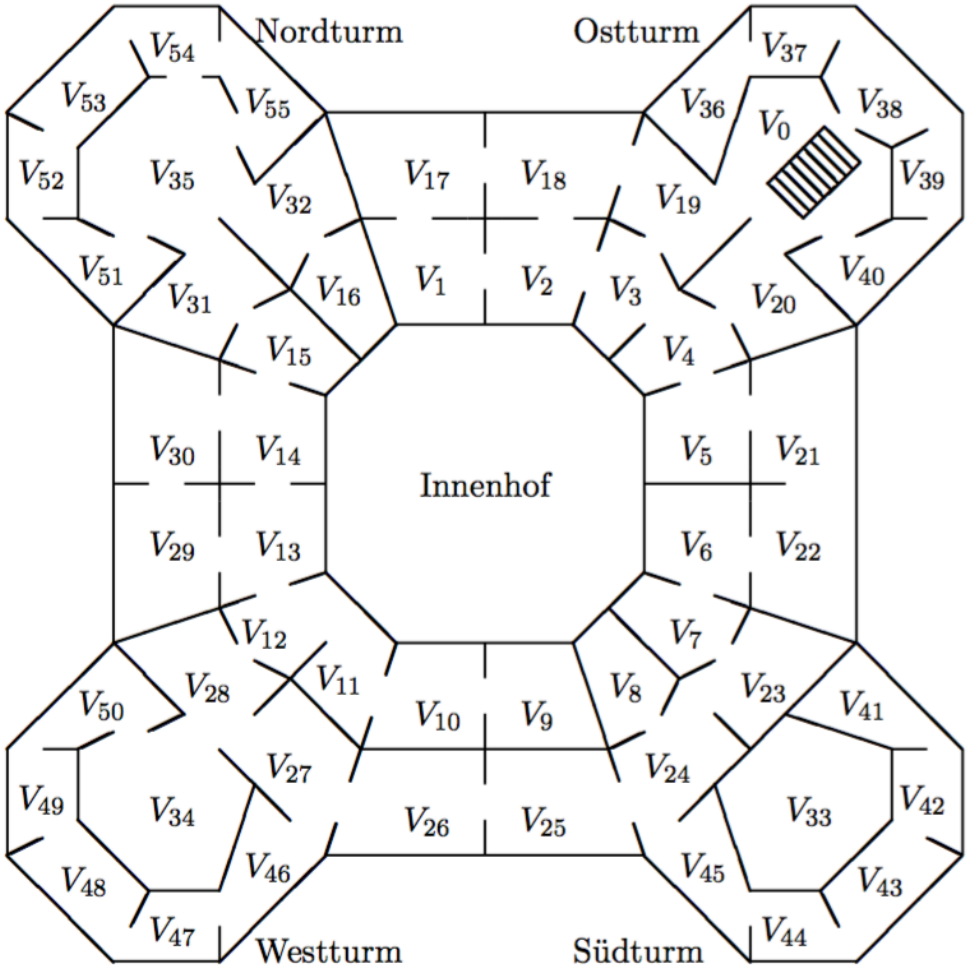
William erinnert sich

„Um den Ausgang aus einem Labyrinth zu finden,“, dozierte William, „gibt es nur ein Mittel. An jedem Kreuzungspunkt wird der Durchgang, durch den man gekommen ist, mit drei Zeichen markiert. Erkennt man an den bereits vorhandenen Zeichen auf einem der Durchgänge, dass man an der betreffenden Kreuzung schon einmal gewesen ist, bringt man an dem Durchgang, durch den man gekommen ist, nur ein Zeichen an. Sind alle Durchgänge schon mit Zeichen versehen, so muss man umkehren und zurückgehen. Sind aber einer oder zwei Durchgänge der Kreuzung noch nicht mit Zeichen versehen, so wählt man einen davon und bringt zwei Zeichen an. Durchschreitet man einen Durchgang, der nur ein Zeichen trägt, so markiert man ihn mit zwei weiteren, so dass er nun drei Zeichen trägt. Alle Teile des Labyrinthes müßten durchlaufen worden sein, wenn man, sobald man an eine Kreuzung gelangt, niemals den Durchgang mit drei Zeichen nimmt, sofern noch einer der anderen Durchgänge frei von Zeichen ist.“

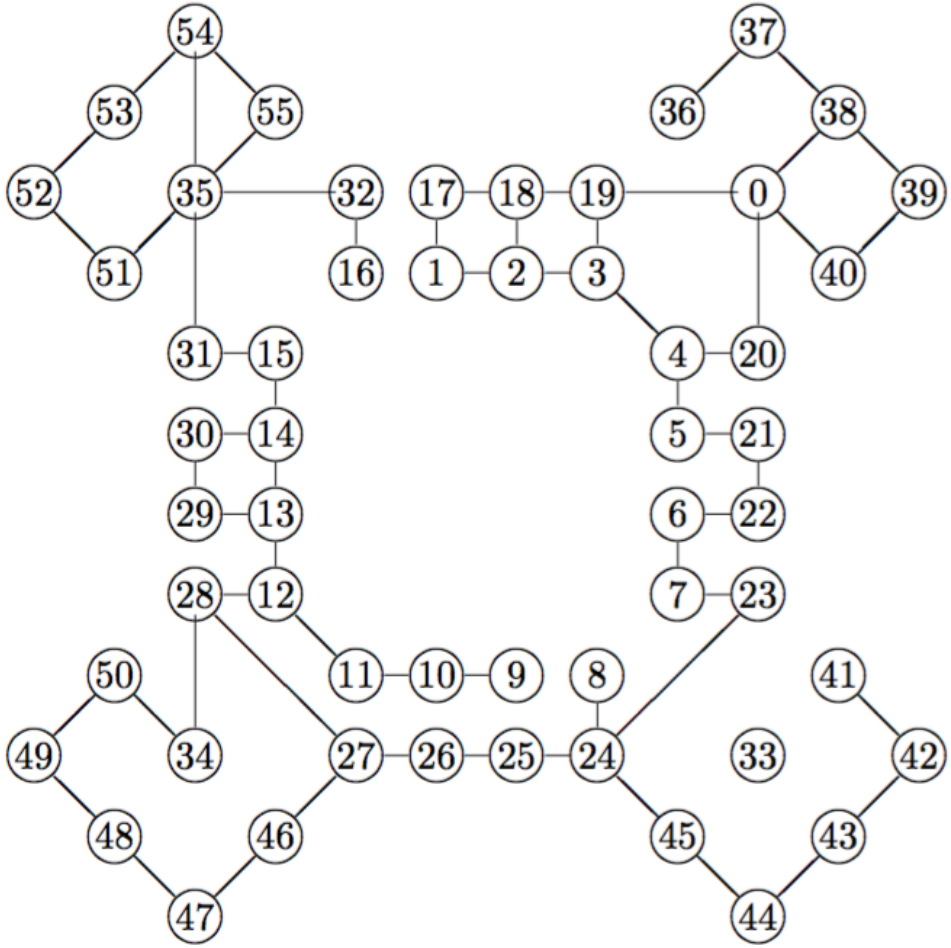
„Woher wißt Ihr das? Seid Ihr ein Experte in Labyrinthen?“
„Nein, ich rezitiere nur einen alten Text, den ich einmal gelesen habe.“
„Und nach dieser Regel gelangt man hinaus?“
„Nicht dass ich wüßte. Aber wir probieren es trotzdem.[...]“

„Der Name der Rose“: Das Labyrinth

Kann man vom Treppenaufgang V_0 aus alle Räume V_i besuchen?



Das Labyrinth als ungerichteter Graph



Der Ariadne-Faden

Geht das denn nicht viel einfacher?

1. Prinzessin Ariadne, Tochter des Königs Minos, hat Theseus den „Ariadne-Faden“ geschenkt, um den Minotauros in einem Labyrinth aufzuspüren und danach wieder aus dem Labyrinth herauszufinden.
2. Theseus hat den Ariadne-Faden während der Suche im Labyrinth abgerollt.
   - Nachdem er den Minotauros getötet hat, braucht er nur den Faden zurückverfolgen, um das Labyrinth wieder verlassen zu können.
3. Aber auch Präorder benutzt den Ariadne-Faden. Und wie, bitte schön, **durchsucht** man das Labyrinth systematisch mit Hilfe eines Fadens?

Tiefensuche: Farbeimer + Ariadne-Faden

Der Algorithmus „**Tiefensuche**“ implementiert und erweitert die Methode des Ariadne-Fadens.

1. Der ungerichtete Graph $G = (V, E)$ und ein Startknoten $s \in V$ ist gegeben.
2. Ganz zu Anfang sind alle Knoten „unmarkiert“. Wir besuchen und **markieren** s.
   // Wir besuchen stets nur **unmarkierte** Knoten.
3. Wenn wir den Knoten u besuchen, betrachten wir nacheinander alle Nachbarn v von u in irgendeiner Reihenfolge.
   - Wenn v markiert ist, tun wir nichts.
   - Wenn v unmarkiert ist, besuchen und **markieren** wir v. Dann wiederholen wir unser Vorgehen rekursiv (für alle mit v benachbarten Knoten).

   Wenn schließlich alle Nachbarn von v markiert sind, dann kehren wir zu u zurück.

(Wir benutzen den Ariadne-Faden und einen Farbeimer.)

Tiefensuche(): Die globale Struktur

Im Array besucht wird vermerkt, welche Knoten bereits besucht wurden.

```
void Tiefensuche()
  {for (int k = 0; k < n; k++) besucht[k] = 0;
    for (k = 0; k < n; k++)
      if (! besucht[k]) tsuche(k); }
```

- Jeder Knoten wird besucht, aber tsuche(v) wird nur dann aufgerufen, wenn v nicht als „besucht“ markiert ist.
- Wie funktioniert tsuche(v)?

tsuche()

- Der gerichtete oder ungerichtete Graph G werde durch seine Adjazenzliste A repräsentiert.
- Die Adjazenzlisten werden definiert durch Listeneinträge der Form

```
struct Knoten {
  int name;
  Knoten * next; }
```

tsuche(v):

1. Zuerst wird v markiert.
2. Dann rufe tsuche rekursiv für alle unmarkierten Nachbarn/Nachfolger von v auf.

```
void tsuche(int v)
  { Knoten *p ; besucht[v] = 1;
    for (p = A[v]; p !=0; p = p->next)
      if (!besucht [p->name]) tsuche(p->name);
        // Die Kante {v, p->name} heißt eine Baumkante }
```

Ungerichtete Graphen: Baum- und Rückwärtskanten

Der Wald der Tiefensuche: ungerichtete Graphen

Wir veranschaulichen das Vorgehen von Tiefensuche, indem wir die Kanten des Graphen $G = (V, E)$ in zwei Klassen aufteilen, nämlich

- **Baumkanten** und
- **Rückwärtskanten**.

- Eine Kante $\{v, w\} \in E$ des Graphen G heißt eine

  Baumkante,

  falls tsuche(w) in der for-Schleife von tsuche(v) aufgerufen wird oder umgekehrt.
  - Die Baumkanten definieren einen Wald, den wir den

    Wald der Tiefensuche

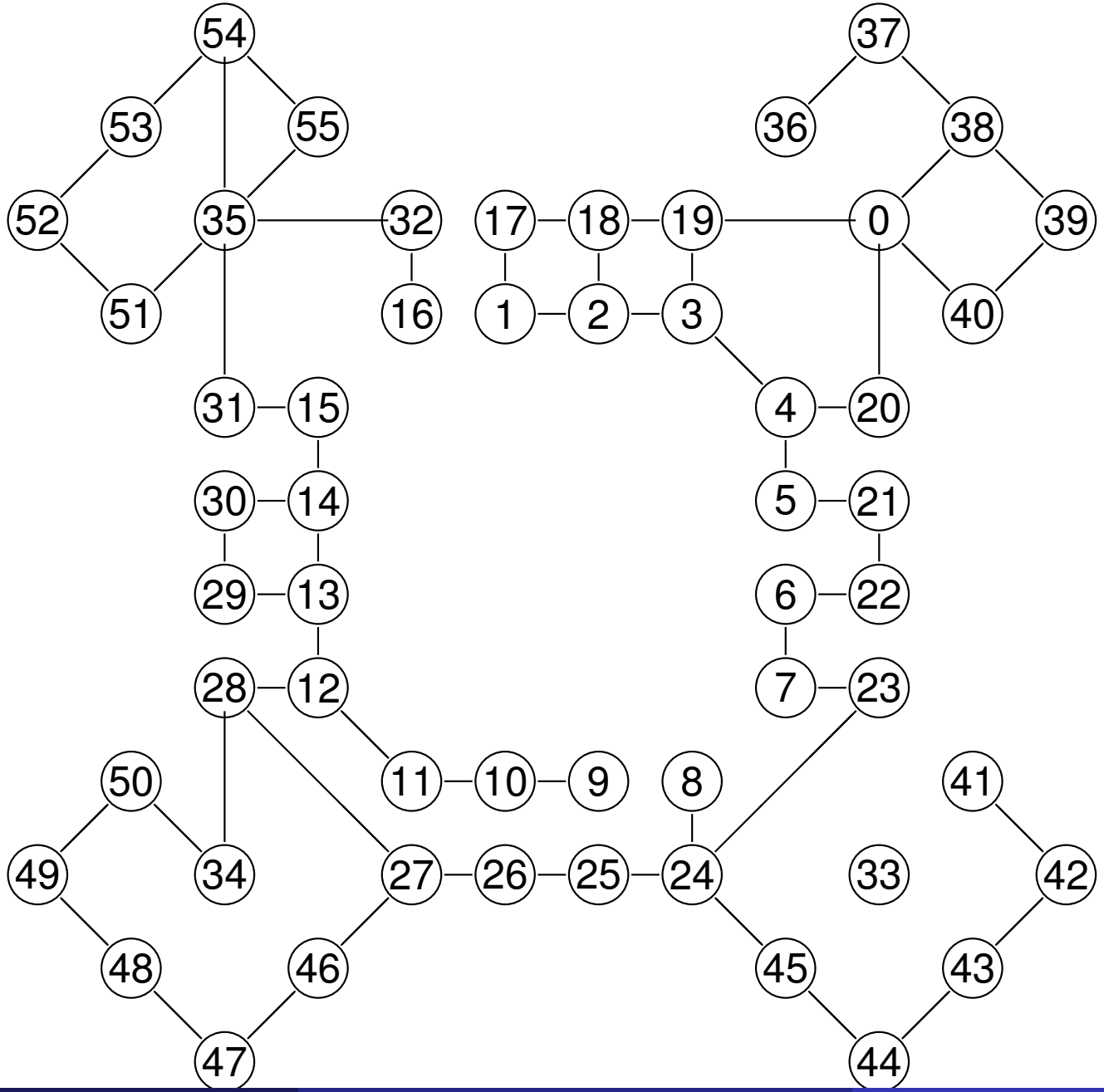
    nennen.
- Eine Kante $\{v, w\} \in E$ heißt eine

  Rückwärtskante,

  falls $\{v, w\}$ keine Baumkante ist.

Warum sprechen wir von Rückwärtskanten?

Tiefensuche: Ein Beispiel



Tiefensuche für ungerichtete Graphen (1/2)

Sei $G = (V, E)$ ein ungerichteter Graph und $\{v, w\}$ sei eine Kante von G.
W_G sei der Wald der Tiefensuche für G.

(a) tsuche(v) werde vor tsuche(w) aufgerufen. Dann ist w ein Nachfahre von v in W_G. Insbesondere gehören v und w zum selben Baum von W_G.

(b) Alle Kanten des Graphen verbinden einen Vorfahren mit einem Nachfahren.

(a) Warum ist w ein Nachfahre von v in W_G?

- tsuche(v) wird vor tsuche(w) aufgerufen:
  Knoten w ist zum Zeitpunkt der Markierung von Knoten v *unmarkiert*.
- tsuche(v) kann nur dann terminieren, wenn w (zwischenzeitlich) markiert wird: w muss während der Ausführung von tsuche(v) markiert werden.

(b) O.B.d.A. werde tsuche(v) vor tsuche(w) aufgerufen. Dann ist w Nachfahre von v:
Die Graphkante $\{v, w\}$ ist eine Baumkante oder eine Rückwärtskante:
In beiden Fällen wird ein Vorfahre mit einem Nachfahren verbunden.

Tiefensuche für ungerichtete Graphen (2/2)

Tiefensuche besucht jeden Knoten genau einmal.

- Das Programm Tiefensuche wird von einer for-Schleife gesteuert, die tsuche(v) für alle noch nicht besuchten Knoten v aufruft.
- Wenn aber tsuche(v) aufgerufen wird, dann wird v sofort markiert: Nachfolgende Besuche sind ausgeschlossen.

Der Baum von v in W_G enthält genau die Knoten der **Zusammenhangskomponente** von v: Die Bäume von W_G entsprechen den Zusammenhangskomponenten von G.

- T sei ein Baum im Wald W_G und T besitze v als Knoten.
  - v erreicht jeden Knoten in T, denn der Baum T ist zusammenhängend: Die Zusammenhangskomponente von v enthält alle Knoten in T.
- Wenn $v = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_m = u$ ein Weg in G ist, dann gehören $v_0, v_1, \ldots, v_m$ alle zum selben Baum:
  Die Knotenmenge von T enthält die Zusammenhangskomponente von v.

Labyrinthe

Tiefensuche löst jedes Labyrinth-Problem, das sich als ungerichteter Graph interpretieren läßt.

- Wenn es möglich ist, von irgendeinem Punkt p aus den Ausgang zu erreichen, dann befinden sich p und Ausgang in derselben Zusammenhangskomponente.
- Der **Tiefensuchbaum** von p wird uns stets einen Weg aus dem Labyrinth zeigen.

Wie schnell findet man aus einem Labyrinth heraus? D.h., wie schnell ist Tiefensuche?

Die Laufzeit von Tiefensuche

Tiefensuche terminiert nach höchstens $O(n + |E|)$ Schritten.

- Zuerst muss der Aufwand für die for-Schleife in Tiefensuche bestimmt werden: $O(n)$ Schritte.
- Wieviele Schritte werden direkt von tsuche(v) ausgeführt
  (und nicht in nachfolgenden rekursiven Aufrufen)?
  $O(\text{grad}(v))$ Operationen, wobei grad(v) die Anzahl der Nachbarn von v ist.
- Wieviele Operationen werden insgesamt ausgeführt?

$$\begin{aligned} O\left(\sum_{v \in V}(1 + \text{grad}\,(v))\right) &= O\left(\sum_{v \in V} 1 + \sum_{v \in V} \text{grad}\,(v)\right) \\ &= O\left(|V| + |E|\right). \end{aligned}$$

Tiefensuche ist sehr schnell.

Tiefensuche und Darstellung multiple choice

Wie lange dauert eine Tiefensuche in einem Graphen mit n Knoten und m Kanten, je nach Darstellungsform des Graphen?

- $m = \Theta(n)$, Adjanzenzliste: (1) $\Theta(n+m)$ (2) $\Theta(n^2)$
- $m = \Theta(n)$, Adjanzenzmatrix: (3) $\Theta(n+m)$ (4) $\Theta(n^2)$
- $m = \Theta(n^2)$, Adjanzenzliste: (5) $\Theta(n+m)$ (6) $\Theta(n^2)$
- $m = \Theta(n^2)$, Adjanzenzmatrix: (7) $\Theta(n+m)$ (8) $\Theta(n^2)$

Auflösung: (1), (4), (5), (6), (7), (8)

Anwendungen der Tiefensuche

Sei $G = (V, E)$ ein ungerichteter Graph. Dann kann in Zeit $O(|V| + |E|)$ überprüft werden, ob

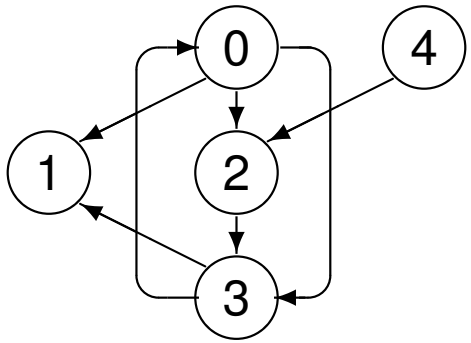
(a) G **zusammenhängend** ist:

- G ist genau dann zusammenhängend, wenn G genau eine Zusammenhangskomponente hat.
- Die Bäume von W_G entsprechen den Zusammenhangskomponenten von G.
- G ist genau dann zusammenhängend, wenn W_G aus genau einem Baum besteht.

(b) G ein **Wald** ist:

- G ist genau dann ein Wald, wenn G keine Rückwärtskanten hat.
- Überprüfe für jede Kante $\{v, w\}$, ob entweder tsuche(w) direkt in tsuche(v) aufgerufen wird oder ob tsuche(v) direkt in tsuche(w) aufgerufen wird.

Tiefensuche für gerichtete Graphen (1/2)



Wenn Tiefensuche im Knoten 0 beginnt und die Knoten aufsteigend in jeder Liste aufgeführt sind:

- Die Kanten $(0, 1)$, $(0, 2)$ und $(2, 3)$ sind **Baumkanten**.
- Die Kante $(3, 0)$ ist eine **Rückwärtskante**.
- Die Kante $(0, 3)$ ist eine **Vorwärtskante**, sie verbindet einen Knoten mit einem Nachfahren, der kein Kind ist.
- Die Kanten $(3, 1)$ und $(4, 2)$ sind **Querkanten**, sie verbinden zwei Knoten, die nicht miteinander „verwandt“ sind.

Gerichtete Graphen:

Baum-, Rückwärts und Vorwärtskanten sowie Rechts-nach-Links Querkanten

Tiefensuche für gerichtete Graphen (2/2)

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste vorliegt.

(a) Tiefensuche besucht jeden Knoten genau einmal.

(b) Die Laufzeit von Tiefensuche() ist durch $O(|V| + |E|)$ beschränkt.

(c) Während der Ausführung von tsuche(v) wird ein Knoten w genau dann besucht, wenn w auf einem Weg liegt,

dessen Knoten vor Beginn von tsuche(v) unmarkiert sind.

- Zu Beginn von tsuche sei w von v aus durch einen „unmarkierten Weg“ erreichbar.
- Dann kann tsuche(v) nur dann terminieren, wenn w während der Ausführung von tsuche(v) markiert wird.

Die folgende Aussage ist falsch: Während der Ausführung von tsuche(v) werden genau die Knoten besucht, die auf einem Weg mit Anfangsknoten v liegen.

Kantentypen für gerichtete Graphen

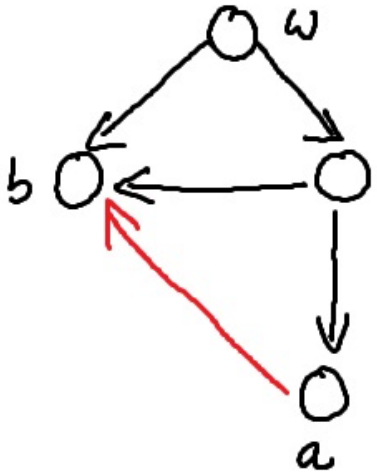
- Es ist nicht verwunderlich, dass durch die Kantenrichtungen neben **Rückwärtskanten** jetzt auch **Vorwärtskanten** vorkommen.
- **Querkanten** sind ein gänzlich neuer Kantentyp.
  - Ein Querkante heißt eine **rechts-nach-links Querkante**, wenn sie von einem **später** besuchten zu einem **früher** besuchten Knoten führt.

Es gibt nur rechts-nach-links Querkanten.

Warum? Sei $e = (v, w)$ eine beliebige Kante.

- Wenn tsuche(v) terminiert, dann ist w markiert.
- Wenn w vor dem Aufruf von tsuche(v) markiert wurde, dann ist e entweder eine Rückwärtskante oder eine rechts-nach-links Querkante.
- Wenn w während des Aufrufs markiert wird, dann ist e entweder eine Vorwärtskante oder eine Baumkante.

Kantenklassifizierung multiple choice



Betrachte die möglichen Abläufe der Tiefensuche mit Startknoten w.

Je nach Sortierung der Nachbarknoten wird Kante (a, b) zu einer

- (1) Baumkante
- (2) Rückwärtskante
- (3) Vorwärtskante
- (4) Querkante

Auflösung: (1), (4)

Eine automatische Erkennung der Kantentypen

Wir benutzen zwei integer-Arrays „**Anfang**“ und „**Ende**“ als Uhren, um den Zeitpunkt des Beginns und des Endes des Besuchs festzuhalten.

```
Anfangnr=Endenr=0;
void tsuche(int v)
  {Knoten *p; Anfang[v] = ++Anfangnr;
    for (p = A[v]; p != 0; p = p->next)
      if (!Anfang[p->name]) tsuche(p->name);
    Ende[v] = ++Endenr; }
```

- $e = (v, w)$ ist eine **Vorwärtskante** $\Leftrightarrow$
  Anfang$[v] <$ Anfang$[w]$ und $e = (v, w)$ ist keine Baumkante.
- $e = (v, w)$ ist eine **Rückwärtskante** $\Leftrightarrow$
  Anfang$[v] >$ Anfang$[w]$ und Ende$[v] <$ Ende$[w]$.
- $e = (v, w)$ ist eine **Querkante** $\Leftrightarrow$
  Anfang$[v] >$ Anfang$[w]$ und Ende$[v] >$ Ende$[w]$.

Kantenklassifizierung multiple choice

Betrachte die möglichen Abläufe der Tiefensuche mit Startknoten w.

Es gilt also immer Anfang[w] = 1.

Welche Werte können bei Anfang[v] auftreten, je nach Sortierung der Nachbarknoten?

Auflösung: Anfang[v] $\in \{3, 4\}$

Kantenklassifizierung demogr.

Für die Kante (v, w) gilt

- Anfang[v] = 3, Ende[v] = 1
- Anfang[w] = 1, Ende[w] = 3

Diese Kante ist eine

- (1) Baumkante
- (2) Rückwärtskante
- (3) Vorwärtskante
- (4) Querkante

Auflösung: (2) Rückwärtskante

Anwendungen der Tiefensuche (1/2)

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste repräsentiert ist.
Dann lassen sich die folgenden Probleme in Zeit $O(|V| + |E|)$ lösen:

(a) Ist G azyklisch?

- Jede Rückwärtskante schließt einen Kreis.
- Baum-, Vorwärts- und Querkanten allein können keinen Kreis schließen.
- G ist azyklisch genau dann, wenn G keine Rückwärtskanten hat.

(b) Führe eine topologische Sortierung durch.

- Führe eine Tiefensuche durch.
- G muß azyklisch sein, hat also nur Baum-, Vorwärts- und rechts-nach-links Querkanten.
- „Sortiere“ die Knoten absteigend nach ihrem Endewert:
  - ⋆ keine Kante führt von einem Knoten mit kleinem Endewert zu einem Knoten mit großem Endewert.

Anwendungen der Tiefensuche (2/2)

G ist **stark zusammenhängend**, wenn es für jedes Knotenpaar (u, v) einen Weg von u nach v gibt.

(c) Ist G stark zusammenhängend?

Es genügt zu zeigen, dass alle Knoten von Knoten 1 aus erreichbar sind und dass jeder Knoten auch Knoten 1 erreicht.

- Alle Knoten sind genau dann von Knoten 1 aus erreichbar, wenn während der Ausführung von tsuche(1) alle Knoten besucht werden.
- Kann jeder Knoten den Knoten 1 erreichen?
  - Kehre die Richtung aller Kanten um,
  - führe tsuche(1) auf dem neuen Graphen aus
  - und überprüfe, ob alle Knoten besucht wurden.

Breitensuche
und die Bestimmung kürzester Wege

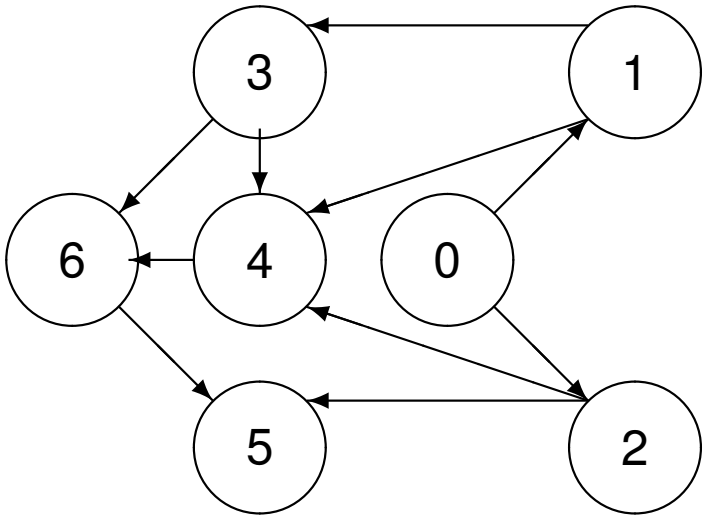
Breitensuche

Breitensuche für einen Knoten v soll zuerst v, dann die „Kindergeneration von v“, gefolgt von den „Enkelkindern“ und den „Urenkeln“ von v .... besuchen.

```
void Breitensuche(int v)

{Knoten *p; int w; queue q;

for (int k =0; k < n ; k++) besucht[k] = 0;
q.enqueue(v); besucht[v] = 1;

while (!q.empty ( ))
  {w = q. dequeue ( );
  for (p = A[w]; p != 0; p = p->next)
    if (!besucht[p->name])
      {q.enqueue(p->name); besucht[p->name] = 1;
      // (w,p->name) ist eine Baumkante. }}}
```

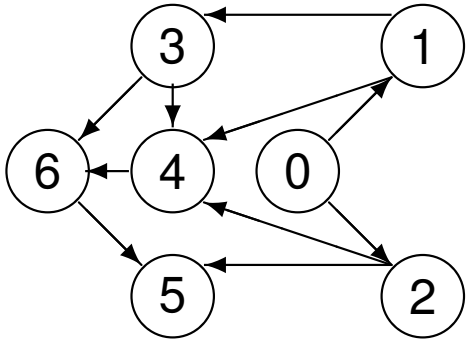
Ein Beispiel

Breitensuche(v) berechnet einen Baum mit Wurzel v, wenn wir alle Baumkanten einsetzen.



Wir beginnen sowohl Tiefensuche wie auch Breitensuche im Knoten 0. Wie sehen der Baum der Tiefensuche und der Baum der Breitensuche aus?

Baumkanten in der Breitensuche multiple choice



Betrachte die möglichen Abläufe der Breitensuche mit Startknoten 0.

Welche der folgenden Kanten ist dann niemals eine Baumkante – also für keine einzige Sortierung der Nachbarn?

- (1) Kante (1,4)
- (2) Kante (2,4)
- (3) Kante (3,4) $\surd$
- (4) Kante (3,6)
- (5) Kante (4,6)
- (6) Kante (6,5) $\surd$

Eigenschaften der Breitensuche (1/2)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Für Knoten w setze

$$V_w = \{u \in V \mid \text{Es gibt einen Weg von } w \text{ nach } u\}$$

und

$$E_w = \{e \in E \mid \text{beide Endpunkte von } e \text{ gehören zu } V_w\}.$$

(a) Breitensuche(w) besucht jeden Knoten in V_w genau einmal und sonst keinen anderen Knoten.

- Nur bisher nicht besuchte Knoten werden in die Schlange eingefügt, dann aber **sofort** als besucht markiert:
  - ⋆ Jeder Knoten wird höchstens einmal besucht.
- Bevor Breitensuche(w) terminiert, müssen alle von w aus erreichbaren Knoten besucht werden.

(b) Breitensuche(w) benötigt Zeit höchstens $O(|V_w| + |E_w|)$.

- Die Schlange benötigt Zeit höchstens $O(|V_w|)$, da genau die Knoten aus V_w eingefügt werden und zwar genau einmal.
- Jede Kante wird für jeden Endpunkt genau einmal in seiner Adjazenzliste „angefasst“. Insgesamt Zeit $O(|E_w|)$.

Eigenschaften der Breitensuche (2/2)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph.

Ein Baum $T(w)$ mit Wurzel w heißt ein **Baum kürzester Wege** für G, falls

(a) $T(w)$ die Knotenmenge V_w hat und

(b) falls für jeden Knoten $u \in V_w$,
der Weg in $T(w)$ von der Wurzel w nach u ein **kürzester Weg** ist.

- Zuerst werden Knoten u im Abstand 1 von w in die Schlange eingefügt:
  Die Kinder von w in $T(w)$ stimmen mit den Nachbarn von w in G überein.
- Zu jedem Zeitpunkt:
  - Wenn ein Knoten u im Abstand d von der Wurzel aus der Schlange entfernt wird, dann werden noch nicht besuchte Nachbarn von u (im Abstand $d + 1$ von w) eingefügt.
  - Breitensuche baut seinen Baum „Generation für Generation“ auf.

Der Baum der Breitensuche ist ein Baum kürzester Wege.

Breitensuche und kürzeste Wege

Der gerichtete oder ungerichtete Graph $G = (V, E)$ liege als Adjazenzliste vor. Dann können wir in Zeit $O(|V| + |E|)$ kürzeste Wege von einem Knoten w zu allen anderen Knoten bestimmen.

- Breitensuche(w) terminiert in Zeit $O(|V| + |E|)$.
- Der Baum der Breitensuche ist ein Baum kürzester Wege!
- Wir können somit sämtliche kürzesten Wege kompakt als einen Baum darstellen:
  - Implementiere den Baum als Eltern-Array.
  - Wir erhalten für jeden Knoten u einen kürzesten Weg von w nach u durch Hochklettern im Eltern-Array.

Prioritätswarteschlangen und Heaps

Prioritätswarteschlangen

Der abstrakte Datentyp „**Prioritätswarteschlange**“: Füge Elemente (mit Prioritäten) ein und entferne jeweils das Element höchster Priorität.

- Eine Schlange ist eine sehr spezielle Prioritätswarteschlange:
  - Die Priorität eines Elements richtet sich nach dem Zeitpunkt des Einfügens.
- Der abstrakte Datentyp „Prioritätswarteschlange“ umfasst die Operationen
  - **insert**(x,Priorität),
  - **delete**_max(),
  - **change_priority**(wo,Priorität*), wähle Priorität* als neue Priorität
  - und **remove**(wo), entferne das durch wo beschriebene Element.

Wir entwerfen eine geeignete Datenstruktur.

Der Heap

Ein Heap ist ein Binärbaum mit

Heap-Struktur,

der Prioritäten gemäß einer

Heap-Ordnung

abspeichert.

Ein Binärbaum T der Tiefe t hat **Heapstruktur**, wenn:

(a) jeder Knoten der Tiefe höchstens $t-2$ genau 2 Kinder hat,

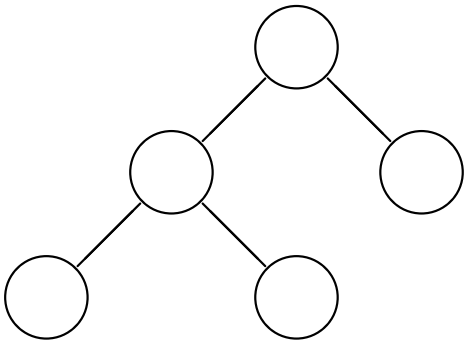
(b) für jeden Knoten v der Tiefe $t-1$ mit weniger als 2 Kindern alle Knoten der Tiefe $t-1$, die rechts von v liegen, keine Kinder haben, und

(c) falls ein Knoten v der Tiefe $t-1$ genau ein Kind hat, dieses Kind ein linkes Kind ist.

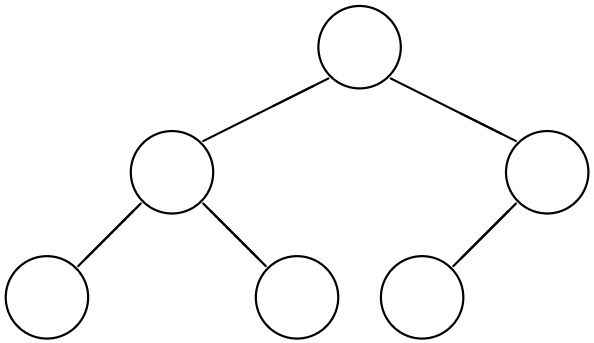
Ein Binärbaum mit Heapstruktur ist ein fast vollständiger binärer Baum: Ist v ein Knoten mit nur einem Kind, so haben alle Knoten links von v zwei Kinder, und alle Knoten rechts von v haben keine Kinder.

Heapstruktur multiple choice

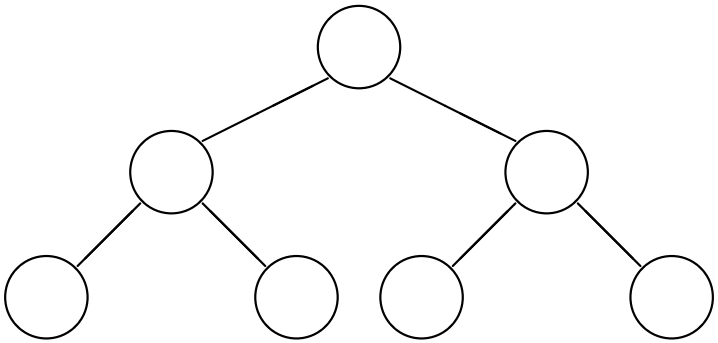
(1)



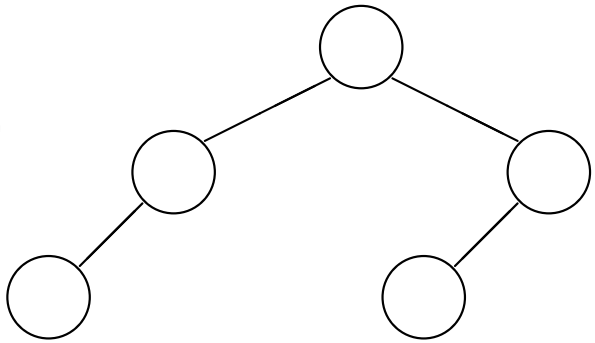
(2)



(3)



(4)



Welcher Baum hat keine Heap-Struktur?

Auflösung: (4)

Heapordnung für Max-Heaps

Ein geordneter binärer Baum T mit Heap-Struktur speichere für jeden Knoten v die Priorität $\mathbf{p(v)}$ von v. Dann hat T

Heap-Ordnung,

falls

$$\mathbf{p(v)} \geq \mathbf{p(w)}$$

für jeden Knoten v und für jedes Kind w von v gilt.

- Die höchste Priorität wird stets an der Wurzel gespeichert.
- Wie sollte man einen Baum mit Heap-Struktur implementieren? Wir arbeiten mit einem Array.

Der Heap

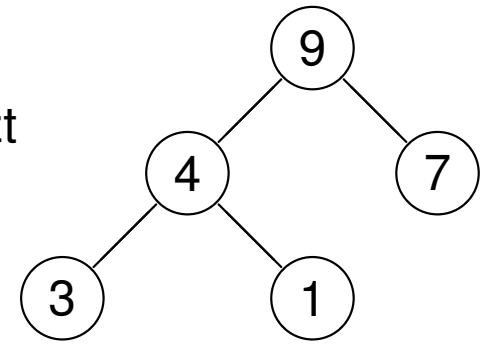
Die Datenstruktur Heap

Der geordnete binäre Baum T habe **Heap-Struktur** und **Heap-Ordnung**.

Das Array H ist ein **Heap** für T, wenn

- $H[1] = p(r)$ für die Wurzel r von T und
- wenn $H[i]$ die Priorität des Knotens v von T speichert, dann gilt
  - $\mathbf{H[2 \cdot i] = p(v_L)}$ für das linke Kind v_L von v und
  - $\mathbf{H[2 \cdot i + 1] = p(v_R)}$ für das rechte Kind v_R.

Zum Beispiel besitzt



den Heap $(9, 4, 7, 3, 1)$.

Die Funktion Insert

Wie **navigiert** man in einem Heap H?

- Wenn Knoten v in Position i gespeichert ist, dann ist
  - das linke Kind v_L in Position $2 \cdot i$,
  - das rechte Kind v_R in Position $2 \cdot i + 1$ und
  - der Elternknoten von v in Position $\lfloor i/2 \rfloor$

  gespeichert.

Wo sollten wir eine neue Priorität p einfügen?

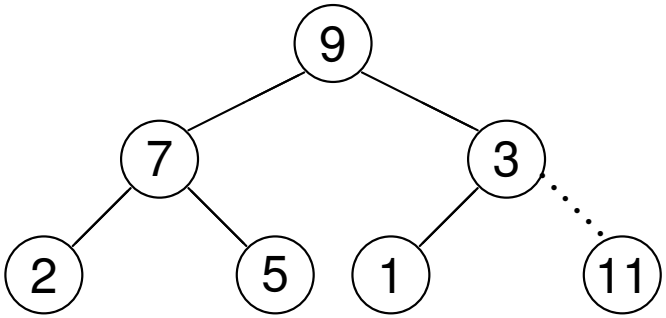
- Es liegt nahe, p auf der ersten freien Position abzulegen. Wir setzen also

$$H[++n] = p.$$

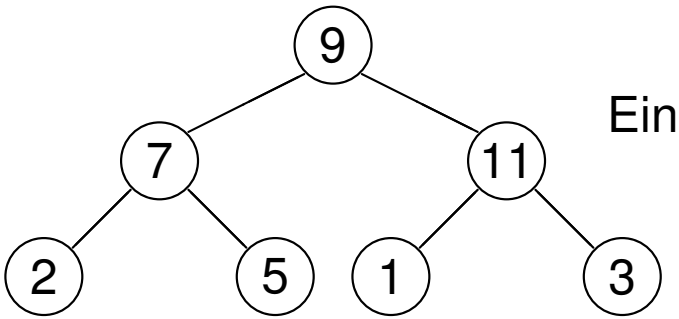
  - Der neue Baum hat Heap-Struktur, aber die Heap-Ordnung ist möglicherweise verletzt.

Wie kann die Heap-Ordnung kostengünstig repariert werden?

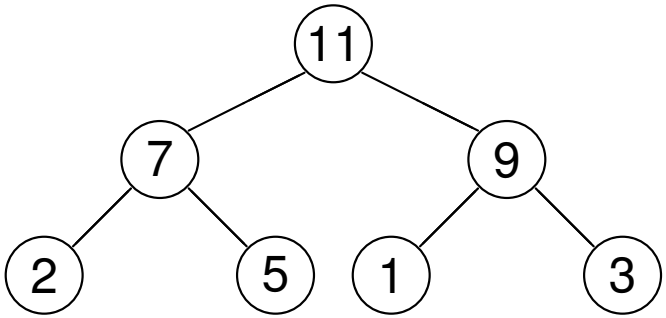
Wir fügen die Priorität 11 ein



Die Heap-Ordnung ist verletzt und 11 rutscht nach oben:



Ein weiterer Vertauschungsschritt repariert die Heap-Ordnung



Repair_up

Die Repair_up Prozedur

Die Klasse heap enthalte die Funktion **repair_up**.

```
void heap::repair_up (int wo)
  {int p = H[wo];
  while ((wo > 1) && (H[wo/2] < p))
      {H[wo] = H[wo/2];
      wo = wo/2; }
  H[wo] = p;}
```

- Wir verschieben die Priorität solange nach oben, bis
  - entweder die Priorität des Elternknotens mindestens so groß ist
  - oder wir die Wurzel erreicht haben.
- Wie groß ist der Aufwand?

Aufwand von Repair_up demogr.

Gegeben sei ein Heap mit n Elementen und Tiefe $t(n)$.

Dann benötigt Repair_up im Worst-Case die Zeit

- (1) $\Theta(1)$
- (2) $\Theta(\log t(n))$
- (3) $\Theta(\log \log n)$
- (4) $\Theta(t(n))$
- (5) $\Theta(\log^5 n + t(n))$
- (6) $\Theta(n + t(n))$

Auflösung: (4) $\Theta(t(n))$

Die Funktion Delete_max()

H repräsentiere einen Heap mit *n* Prioritäten. Für Delete_max:

1. Überschreibe die Wurzel mit $H[n]$
2. und verringere *n* um 1.

- Durch das Überschreiben mit $H(n)$ ist das entstandene Loch an der Wurzel verschwunden:

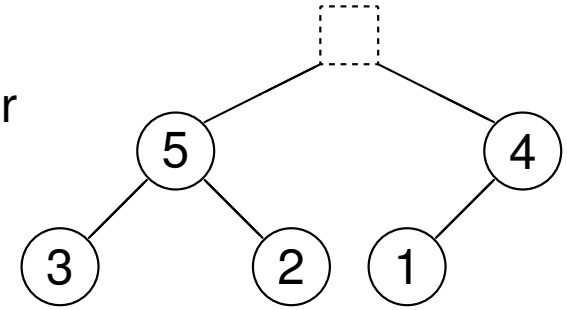
  Die Heap-Struktur ist wiederhergestellt.

- Allerdings ist die Heap-Ordnung möglicherweise verletzt und muss repariert werden.

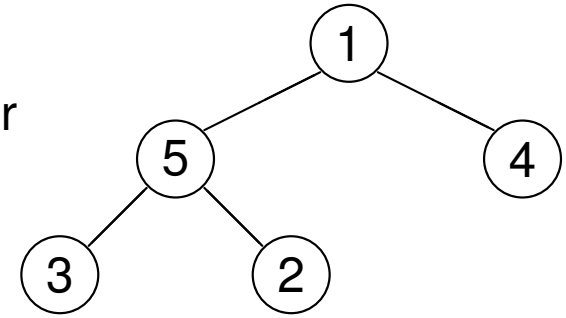
  Die Prozedur **repair_up** versagt: sie ist nur anwendbar, wenn die falsch stehende Priorität größer als die Eltern-Priorität ist.

Ein Beispiel

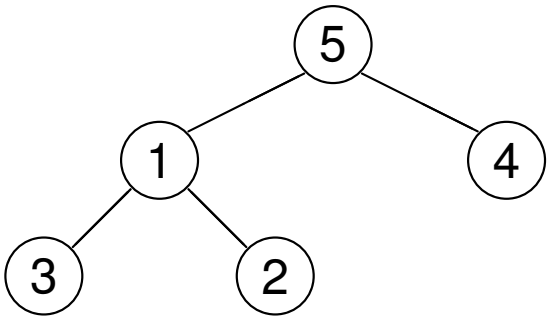
Vorher



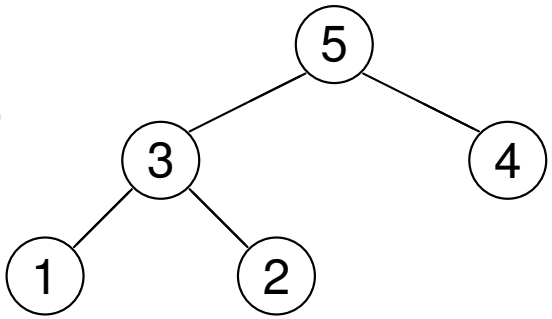
und nachher



Repariere die Heap-Ordnung: Vertausche mit dem **größtem** Kind



und wiederhole



und fertig.

Repariere die Heap-Ordnung nach unten.

Repair_down

Die Prozedur Repair_down

Die Klasse heap enthalte die Funktion **repair_down**.

```
void heap::repair_down (int wo)
  {int kind; int p = H[wo];
  while (wo <= n/2)
      {kind = 2 * wo;
      if ((kind < n) && (H[kind] < H[kind + 1])) kind ++;
      if (p >= H[kind]) break;
      H[wo] = H[kind]; wo = kind; }
  H[wo] = p; }
```

- Die Priorität *p* wird mit der Priorität des „größten Kinds“ verglichen und möglicherweise vertauscht.
- Die Prozedur endet, wenn wo die richtige Position ist, bzw. wenn wo ein Blatt beschreibt.
- Wie groß ist der Aufwand? Höchstens proportional zur Tiefe.

Change_priority und Remove

- **void change_priority** (int *wo*, int *p*):
  - Wir aktualisieren die Priorität, setzen also H[*wo*] $= p$.
  - Aber wir verletzen damit möglicherweise die Heap-Ordnung!
    - ⋆ Wenn die Priorität angewachsen ist, dann rufe **repair_up** auf.
    - ⋆ Ansonsten hat sich die Priorität verringert und **repair_down** ist aufzurufen.
- **void remove**(int *wo*):
  - Stelle die Heap-Struktur durch H[*wo*] $=$ H[*n*- -]; wieder her
  - und rufe dann change_priority auf.

Alle vier Operationen

insert, delete_max, change_priority und **remove**

benötigen Zeit höchstens proportional zur Tiefe des Heaps.

Die Tiefe eines Heaps mit n Knoten

Der Binärbaum T besitze Heap-Struktur.

- Wenn T die Tiefe $t = \mathit{Tiefe}(T)$ besitzt, dann hat T mindestens

$$1 + 2^1 + 2^2 + \ldots + 2^{t-1} + 1 = 2^t$$

  Knoten
- aber nicht mehr als

$$1 + 2^1 + 2^2 + \ldots + 2^{t-1} + 2^t = 2^{t+1} - 1$$

  Knoten.
- Also folgt für die Knotenzahl n,

$$2^{\mathit{Tiefe}(T)} \leq n < 2^{\mathit{Tiefe}(T)+1}.$$

Es ist

$$\mathit{Tiefe}(T) = \lfloor \log_2 n \rfloor$$

und alle vier Operationen werden somit in logarithmischer Zeit unterstützt!

Heapsort

Ein Array $(A[1], \ldots, A[n])$ ist zu sortieren.

```
for (i=1; i <= n ; i++)
    insert(A[i]);
// buildheap ist schneller
int N = n;
for (n=N; n >= 1 ; n- -)
    A[n] = delete_max( );
//Das Array A ist jetzt aufsteigend sortiert.
```

- Zuerst werden n Schlüssel eingefügt und dann wird n Mal das Maximum entfernt.
- Sowohl die anfängliche Einfügephase wie auch die letztliche Entfernungsphase benötigen Zeit höchstens $O(n \cdot \log_2 n)$.

Heapsort ist eines der schnellsten Sortierverfahren.

Wie kann der Heap schneller geladen werden?

Führe statt vielen kleinen Reparaturen eine große Reparatur durch.

- Beginne die Reparatur mit den Blättern. Jedes Blatt ist schon ein Heap und eine Reparatur ist nicht notwendig.
- Wenn t die Tiefe des Heaps ist, dann kümmern wir uns als Nächstes um die Knoten v der Tiefe $t - 1$.
  - Sei T_v der Teilbaum mit Wurzel v.
  - T_v ist nur dann kein Heap, wenn die Heap-Ordnung im Knoten v verletzt ist: Repariere mit repair_down, gestartet in v.
  - Höchstens ein Vertauschungsschritt wird benötigt.
- Wenn v ein Knoten der Tiefe $t - j$ ist, dann muss höchstens die Heap-Ordnung im Knoten v repariert werden.
  - Höchstens j Vertauschungsschritte genügen.

Es gibt nur wenige teure Reparaturschritte!

Analyse

- Es gibt 2^{t-j} Knoten der Tiefe $t-j$ (für $j \geq 1$).
- Für jeden dieser Knoten sind höchstens j Vertauschungsschritte durchzuführen und die Ladezeit ist durch $\sum_{j=1}^{t} j \cdot 2^{t-j}$ beschränkt.
- Behauptung: $\sum_{j=1}^{t} j \cdot 2^{t-j} = 2^{t+1} - t - 2$. Wir geben einen induktiven Beweis:

$$\begin{aligned}\sum_{j=1}^{t+1} j \cdot 2^{t+1-j} &= 2\sum_{j=1}^{t} j \cdot 2^{t-j} + t + 1 \\ &= 2 \cdot (2^{t+1} - t - 2) + t + 1 \\ &= 2^{t+2} - (t+1) - 2.\end{aligned}$$

Der Heap kann in linearer Zeit geladen werden.

Die Klasse heap

```
class heap
{private:
    int *H; // H ist der Heap.
    int n; // n bezeichnet die Größe des Heaps.
    void repair_up (int wo);
    void repair_down (int wo);
public:
    heap (int max) // Konstruktor.
        { H = new int[max]; n = 0; }
    int read (int i) { return H[i]; }
    void insert (int priority);
    int delete_max( );
    void change_priority (int wo, int p);
    void remove(int wo);
    void buildheap();
    void heapsort(); };
```

Prioritätswarteschlangen: Zusammenfassung

(a) Ein Heap mit n Prioritäten unterstützt jede der Operationen

insert, **delete_max**, **change_priority** und **remove**

in Zeit $O(\log_2 n)$.

- Für die Operationen **change_priority** und **remove** muss die Position der zu ändernden Priorität bestimmt werden. (Übungsaufgabe)

(b) **buildheap** baut einen Heap mit n Prioritäten in Zeit $O(n)$.

(c) **heapsort** sortiert n Zahlen in Zeit $O(n \log_2 n)$.

Das kürzeste-Wege Problem und Dijkstra's Algorithmus

Das Single-Source-Shortest Path Problem

Für einen gerichteten Graphen

$$G = (V, E)$$

und eine Längen-Zuweisung

$$\text{länge} : \ E \to \mathbb{R}_{\geq 0}$$

bestimme kürzeste Wege von einem ausgezeichneten Startknoten $s \in V$ zu allen Knoten von G.

Dijkstra's Algorithmus (1/2)

Die Länge eines Weges ist die Summe seiner **Kantengewichte**.

- Mit Breitensuche können wir kürzeste-Wege Probleme lösen, falls länge(e) $= 1$ für jede Kante $e \in E$ gilt.
- Für allgemeine nicht-negative Längen brauchen wir ein stärkeres Geschütz.
- Die **zentrale Beobachtung**.
  - Kantengewichte sind nicht-negativ: Die kürzeste, mit einem Knoten v inzidente Kante (v, w) ist ein kürzester Weg von v nach w.
  - Dijkstra's Algorithmus setzt diese Beobachtung wiederholt ein.

Dijkstra's Algorithmus (2/2)

(1) Setze $S = \{s\}$ und

$$\text{distanz}[v] = \begin{cases} \text{länge}\ (s, v) & \text{wenn} \quad (s, v) \in E \\ \infty & \text{sonst.} \end{cases}$$

/* distanz[v] ist die Länge des bisher festgestellten kürzesten Weges von s nach v. */

(2) Solange $S \neq V$ wiederhole

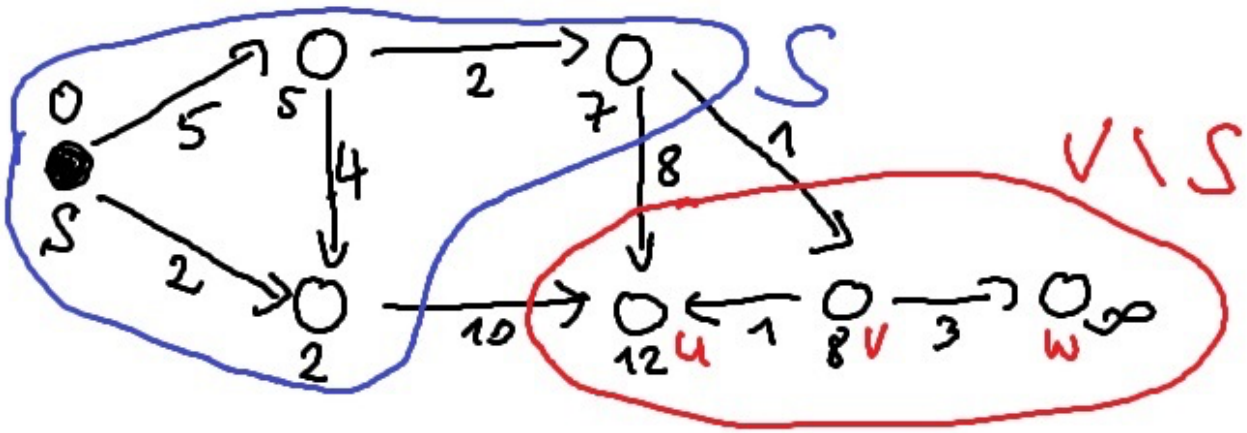
(a) wähle einen Knoten $w \in V \setminus S$ mit kleinstem Distanz-Wert.
/* distanz[w] ist die tatsächliche Länge eines kürzesten Weges von s nach w. **Warum? GL-1** */

(b) Füge w in S ein.

(c) Aktualisiere die Distanz-Werte der Nachfolger von w:
Setze für jeden Nachfolger $u \in V \setminus S$ von w

- c = distanz[w] + länge(w,u);
- distanz[u] = (distanz[u] > c) ? c : distanz[u];

Dijkstra Ablauf demogr.



In welcher Reihenfolge werden die restlichen Knoten in S aufgenommen?

- (1) u, v, w
- (2) v, u, w
- (3) w, v, u
- (4) u, w, v
- (5) w, u, v
- (6) v, w, u

Auflösung: (2) v, u, w

Datenstrukturen für Dijkstra's Algorithmus

In der Vorlesung „Theoretische Informatik 1 (GL-1)“ wird gezeigt, dass Dijkstra's Algorithmus korrekt ist und das kürzeste-Wege Problem effizient löst.

Welche Datenstrukturen sollten wir einsetzen?

- Darstellung des Graphen G:
  - Wir implementieren G als **Adjazenzliste**, da wir dann sofortigen Zugriff auf die Nachfolger u von w im Aktualisierungschritt (2c) haben.
- Implementierung der Menge $V \setminus S$:
  - Ein Knoten w mit kleinstem Distanzwert ist zu bestimmen und zu entfernen.
  - Knoten sind gemäß ihrem anfänglichen Distanzwert einzufügen.

  Wähle einen **Min-Heap** für $V \setminus S$:
  - Ersetze die Funktion delete_max() durch die Funktion delete_min().
  - Implementiere den Aktualisierungschritt (2c) durch change_priority(wo, c). Woher kennen wir die Position wo?

Minimale Spannbäume

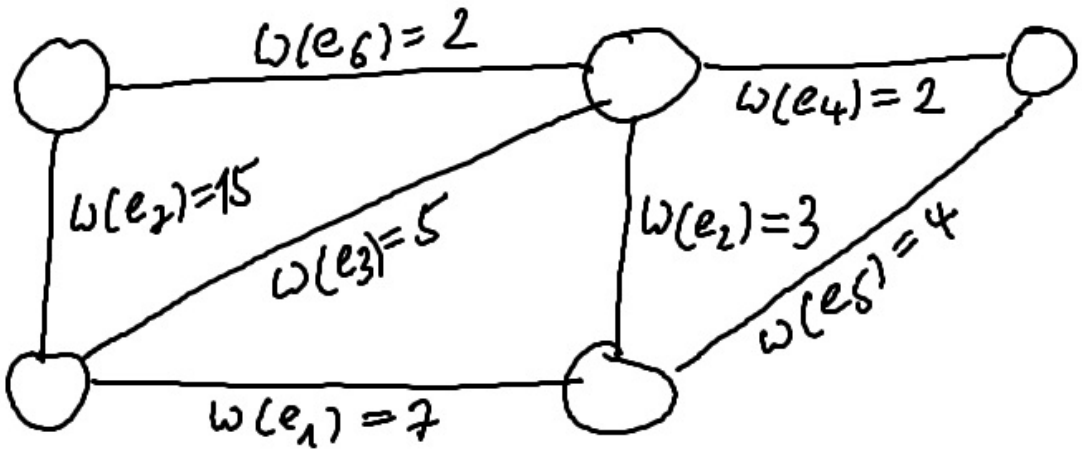
Minimale Spannbäume

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph.
Jede Kante $e \in E$ erhält eine reelwertige Länge „länge(e)“.

- Ein Baum $T = (V', E')$ heißt ein Spannbaum für G, falls $V' = V$ und $E' \subseteq E$.
- Die Länge eines Spannbaums ist die Summe der Längen seier Kanten.
- Ein minimaler Spannbaum ist ein Spannbaum minimaler Länge.

- Je zwei Knoten von G bleiben auch in einem Spannbaum miteinander verbunden, denn ein Baum ist zusammenhängend. Wenn wir aber irgendeine Kante entfernen, dann zerstören wir den Zusammenhang.
- Wenn alle Kantenlängen nicht-negativ sind, dann suchen wir nach einem zusammenhängenden Teilgraph von G minimaler Länge.

Minimale Spannbäume Beispiel demogr.



Welche Kanten sind nicht im minimalen Spannbaum?

Auflösung:

Der Algorithmus von Prim: Die Idee

- Angenommen wir wissen, dass ein Baum B in einem minimalen Spannbaum enthalten ist.
- Wir möchten eine kreuzende Kante zu B hinzufügen: e soll also einen Knoten in B mit einem Knoten außerhalb von B verbinden.

- Der Algorithmus von Prim wählt eine kürzeste kreuzende Kante.
- In der Vorlesung „Algorithmentheorie“ wird gezeigt, dass auch $B \cup \{e\}$ in einem minimalen Spannbaum enthalten ist:
  Der Algorithmus berechnet also einen minimalen Spannbaum.
- Worauf müssen wir bei der Implementierung achten?
  - Eine kürzeste kreuzende Kante muss schnell gefunden werden.
  - Wenn der Baum B um einen neuen Knoten u anwächst, dann erhalten wir neue kreuzende Kanten, nämlich in u endende Kanten.

Der Algorithmus von Prim

(1) Setze $S = \{1\}$.

/* B ist stets ein Baum mit Knotenmenge S. Zu Anfang besteht B nur aus dem Knoten 1.

(2) Solange $S \neq V$, wiederhole:

(a) Bestimme eine kürzeste kreuzende Kante $e = \{u, v\}$.
(b) Füge e zu B hinzu.
(c) Wenn $u \in S$, dann füge v zu S hinzu. Ansonsten füge u zu S hinzu.
/* Beachte, dass wir neue kreuzende Kanten erhalten, nämlich alle Kanten die den neu hinzugefügten Knoten als einen Endpunkt und einen Knoten aus $V \setminus S$ als den anderen Endpunkt besitzen.

Die Datenstruktur für Prim's Algorithmus

- Für jeden Knoten $u \in V \setminus S$ bestimmen wir die Länge $l(u)$ einer kürzesten Kante, die u mit einem Knoten in S verbindet.
- Wir verwalten die Knoten in $V \setminus S$ mit einer Prioritätswarteschlange und definieren $l(u)$ als die Priorität des Knotens u.
  - Initialisiere einen Min-Heap, indem jeder Nachbar u von 1 mit Priorität länge$(\{1, u\})$ einfügt wird, bzw. mit Priorität ∞, wenn u kein Nachbar ist.
  - Wir bestimmen eine kürzeste kreuzende Kante, wenn wir einen Knoten in $u \in V \setminus S$ mit niedrigster Priorität bestimmen.
  - Beachte, dass sich nur die Prioritäten der Nachbarn von u ändern.
- Implementiere G durch eine Adjazenzliste, da wir stets nur auf die Nachbarn eines Knotens zugreifen müssen.

Kruskal's Algorithmus: Die Idee

- Prim lässt einen minimalen Spannbaum „Kante für Kante“ wachsen.
- Kruskal's Algorithmus beginnt mit einem Wald von Einzelknoten.
  - Die Kanten werden nach aufsteigender Länge sortiert und der Reihe nach, beginnend mit Kanten kürzester Länge verarbeitet.
  - Wenn die Kante e „dran“ ist und keinen Kreis schließt, dann wird die Kante zum Wald hinzugefügt.
  - Ansonsten schließt die Kante einen Kreis und wird verworfen.

Wie stellt man fest, ob eine Kante einen Kreis schließt?

Kruskal's Algorithmus

(1) Sortiere die Kanten gemäß aufsteigender Länge. Sei $W = (V, F)$ der leere Wald, also $F = \emptyset$.

(2) Solange W kein Spannbaum ist, wiederhole

  (a) Nimm die gegenwärtig kürzeste Kante e und entferne sie aus der sortierten Folge.
  (b) Verwerfe e, wenn e einen Kreis in W schließt.
  (c) Ansonsten setze $F = F \cup \{e\}$: e wird zum Wald W hinzugefügt.

Wir beschränken uns auf die Implementierung.
In der Vorlesung „Algorithmentheorie“ wird gezeigt, dass Kruskal's Algorithmus korrekt ist.

Die Union-Find Datenstruktur I

- Wir sortieren die Kanten zum Beispiel mit Heapsort.
- Danach müssen wir für alle Kanten $e = \{u, v\}$ entscheiden, ob e einen Kreis in W schließt.
  - Die Operation find(u) bestimme die Wurzel w_u des Baums, der u enthält.
  - Die Kante e schließt genau dann einen Kreis, wenn find(u) = find(v).
- Wenn e keinen Kreis in W schließt, dann müssen wir die Bäume mit den Wurzeln find(u) und find(v) vereinigen. Dazu benutzen wir die Operation union(u, v).

Wie sollten wir den Wald W implementieren?

Die Union-Find Datenstruktur II

Wir implementieren den Wald W durch ein Eltern-Array.

- Zu Anfang ist Eltern$[i] = i$ für alle Knoten i. (Wir fassen i immer dann als eine Wurzel auf, wenn Eltern$[i] = i$ gilt.)
- Wie ist find(u) zu implementieren?
  - Klettere den Baum von u mit Hilfe des Eltern-Arrays hoch.
  - Die „Kletter-Zeit“ ist durch die Tiefe des Baums beschränkt.
  - Wie garantieren wir, dass die Bäume nicht zu tief werden?
- Wenn wir zwei Bäume vereinigen, dann hänge die Wurzel des kleineren Baums unter die Wurzel des größeren Baums!
  - Betrachte einen beliebigen Knoten v.
  - Die Tiefe vergrößert sich nur dann um 1, wenn v dem kleineren Baum angehört. Wenn die Tiefe von v um 1 anwächst, dann wird sich der Baum von v in seiner Größe mindestens verdoppeln.
  - Also ist die Tiefe aller Bäume durch $\log_2(|V|)$ beschränkt.

Die Union-Find Datenstruktur: Das Fazit

Ein union-Schritt benötigt nur konstante Zeit, während ein find-Schritt höchstens logarithmische Zeit benötigt.

- Mit der union-Operation modifizieren wir den minimalen Spannbaum!
- Wir benötigen zwei Datenstrukturen, nämlich
  - die Union-Find Datenstruktur und
  - eine zweite Datenstruktur, die die Kanten des minimalen Spannbaums abspeichert.

Kürzeste Wege vs. Minimale Spannbäume demogr.

Sei G ein Graph mit ganzzahligen Kantengewichten in $\{1, \ldots, n\}$ und

- K die Summe der Kantengewichte in einem Baum kürzester Wege von G
- M die Summe der Kantengewichte in einem minimalen Spannbaum von G

Wie groß kann das Verhältnis K/M werden?

- (1) $\Theta(1)$
- (2) $\Theta(\log n)$
- (3) $\Theta(\sqrt{n})$
- (4) $\Theta(n)$
- (5) $\Theta(n^2)$

Auflösung: (4) $\Theta(n)$