Algorithmen und Datenstrukturen
Kapitel 4
Neue Datenstrukturen, besseres (?) Sortieren

Frank Heitmann
heitmann@informatik.uni-hamburg.de

4. November 2015

Ablauf heute

- Elementare und strukturierte Datentypen
- Datenstrukturen:
  - Stapel (Keller, Stack) und Warteschlangen (Queue)
  - Listen
    - einfach verkettete Listen
    - doppelt verkettete Listen
  - Bäume
  - Graphen
  - Heaps
- HeapSort

Datentypen

Elementare und strukturierte Datentypen

- Elementare Datentypen sind z.B. integer, boolean, char, ...
  - Stehen üblicherweise in einer Programmiersprache zur Verfügung
- Strukturierte Datentypen sind z.B. Arrays und Records, ...
  - "Ansammlung" von elementaren (oder anderen strukturierten) Datentypen.
  - Arrays sind eine (über natürliche Zahlen) indizierte Menge
    - kann man auch als einfache Datenstrukturen sehen.
  - Records in Java:
    - Mehrere Datentypen/Strukturen als Objektvariablen
    - Ggf. getter/setter-methoden
- Datenstrukturen: Daten in einer Struktur abgelegt und Methoden zum Zugriff u.ä.

Stapel (Keller, Stack)

Ein *Stack* verhält sich wie ein Buchstapel.

- Man kann oben - *aber nur oben* - immer weiter drauf legen.
- Man kann von oben - *aber nur von oben* - wieder etwas weg nehmen.
- Man kann stets nur *genau ein Buch* drauf legen oder weg nehmen.

Wichtige Anmerkung

LIFO-Prinzip: Speicherung mit Zugriffsmöglichkeit nur auf dem *zuletzt* gespeicherten Objekt. (*LIFO = last in, first out*)

Stack - Methoden

Ein Stack

- stellt etwas zum Speichern der Daten zur Verfügung (z.B. ein Array)
- implementiert folgende Methoden:
  - *top()/head()* - liefert das oberste Element
  - *push(x)* - legt x auf dem Stack ab
  - *pop()* - liefert und entfernt das oberste Element
  - *isEmpty()* - ist der Stack leer?
  - (*size()* - liefert die Größe des Stacks)
- Implementierung sequentiell oder verkettet ($\Rightarrow$ Liste)

Ablauf

Ein möglicher Ablauf (ganz rechts im Array, ganz oben im Stack)

- Stack zu Anfang: [] (leer)
- isEmpty() - liefert true
- push(3) - Stack: [3]
- isEmpty() - liefert false
- push(5) - Stack: [3,5] (5 oben!)
- top() - liefert 5. Stack: [3,5] (unverändert)
- pop() - liefert 5. Stack: [3] (oberstes Element entfernt!)
- push(7), push(3), push(8) - Stack: [3,7,3,8]

Stack - Interface und Implementation

Stack Interface

- siehe Java/Stack/Stack.java

Stack Implementation

- siehe Java/Stack/ArrStack.java

Und ein kleiner Probelauf

- in Java/Stack/StackTest.java

Anmerkungen

Bei der Implementierung wäre noch auf Grenzfälle zu achten:

1. Was passiert, wenn bei top() oder pop() der Stack leer ist?
2. Was passiert, wenn bei push() der Stack voll ist?

Mögliche Lösungen:

1. Mit if-Klauseln abfangen und mit Fehlermeldungen arbeiten (z.B. null zurückgeben) oder (besser) gleich mit Exceptions. Alternativ: Dokumentieren und die Arbeit dem Aufrufer überlassen.
2. Wie bei 1. oder Platz dynamisch erweitern. Dies geht mit Arrays, ist aber meist umständlich - besser sind dann Listen (s.u.)

Ein Problem

Wie findet man heraus ob ein gegebener String aus öffnenden und schließenden Klammern ein korrekter Klammerausdruck ist?

- (()) ist korrekt
- ()((()()) ist korrekt
-)(ist nicht korrekt
- (()(()))(()) ist nicht korrekt

Eine Lösung

- Öffnende Klammern auf den Stack pushen
- Wenn eine schließende kommt, eine (die zugehörige!) öffnende vom Stack holen.
- Stack muss am Ende leer sein und pop() muss immer möglich sein, wenn wie oben vorgegangen wird.

Zusammenfassung: Stack

Stack

- Zugriff: LIFO
- Operationen: top/head, push, pop, isEmpty, (size)
- Implementierung:
  - Sequentiell (Array)
    - Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Erkennen wohlgeformter Klammerausdrücke
  - Ermitteln der zusammengehoerigen Klammerpaare in $O(n)$ (Idee: Index der öffnenden Klammer auf Stack pushen)
  - Noch schneller geht's ohne Stack indem man einen Zähler bei öffnenden Klammern um eins erhöht und bei schließenden Klammern um eins verringert.

Schlange (Queue)

Eine *Queue* verhält sich wie eine (Warte-)Schlange in einem Supermarkt:

- Man kann sich hinten - *aber nur hinten* - anstellen (hinzufügen).
- Man kann vorne - *aber nur vorne* - Kunden bedienen (wegnehmen).
- Wie beim Stack: Stets nur ein Element hinzufügen/wegnehmen.

Wichtige Anmerkung

FIFO-Prinzip: Speicherung mit Zugriffsmöglichkeit nur auf dem *zuerst* gespeicherten Objekt. (*FIFO = first in, first out*)

Queue - Methoden

Eine Queue

- stellt etwas zum Speichern der Daten zur Verfügung (z.B. ein Array)
- implementiert folgende Methoden:
  - *head()* - liefert das oberste Element
  - *enqueue(x)* - fügt x an das Ende der Schlange ein
  - *dequeue()* - liefert und entfernt das vorderste Element
  - *isEmpty()* - ist die Queue leer?
  - (*size()* - liefert die Größe der Queue)
- Implementierung sequentiell oder verkettet ($\Rightarrow$ Liste)

Ablauf

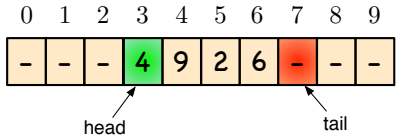
Ein möglicher Ablauf (ganz links im Array, ganz vorne in der Liste)

- Queue zu Anfang: [] (leer)
- isEmpty() - liefert true
- enqueue(3) - Queue: [3]
- isEmpty() - liefert false
- enqueue(5) - Queue: [3,5] (3 vorn!)
- head() - liefert 3. Queue: [3,5] (unverändert)
- dequeue() - liefert 3. Queue: [5] (vorderstes Element entfernt!)
- enqueue(7), enqueue(3), enqueue(8) - Queue: [5,7,3,8]

Ein Array schrumpft und wächst aber nicht so gut! Wie macht man das?

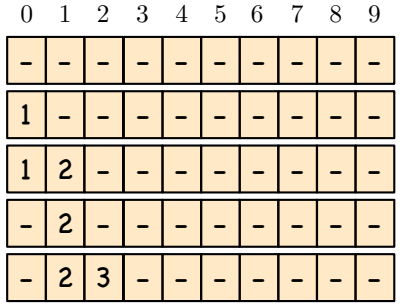
Queue - zyklische Speicherung

Lösung(smöglichkeit): Array als *zyklischen Speicher* benutzen!



- *head* und *tail* "laufen" mit.
- *tail* zeigt auf die erste freie Position am Ende.
- Zeigen *head* und *tail* auf das gleiche Element, ist die Queue leer.
- *Die Queue speichert daher (i.A.) maximal ein Element weniger als die Größe des Arrays!*
- Läuft *tail* "rechts raus", so läuft es "links rein" (modulo Rechnung mit der Größe des Arrays).

Queue - zyklische Speicherung



enqueue(1), enqueue(2), dequeue(1), enqueue(3)

Queue - zyklische Speicherung

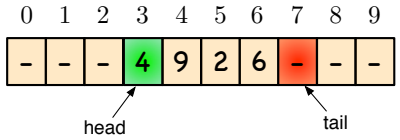
Was passiert, wenn das Array voll wird?

0	1	2	3	4	5	6	7	8	9
-	-	-	-	-	-	-	5	2	-
-	-	-	-	-	-	-	5	2	3
4	-	-	-	-	-	-	5	2	3
4	-	-	-	-	-	-	-	2	3
4	6	-	-	-	-	-	-	2	3

enqueue(3), enqueue(4), dequeue(5), enqueue(6)

Queue - zyklische Speicherung (Wdh.)

Lösung(smöglichkeit): Array als *zyklischen Speicher* benutzen!



- *head* und *tail* "laufen" mit.
- *tail* zeigt auf die erste freie Position am Ende.
- Zeigen *head* und *tail* auf das gleiche Element, ist die Queue leer.
- *Die Queue speichert daher (i.A.) maximal ein Element weniger als die Größe des Arrays!*
- Läuft *tail* "rechts raus", so läuft es "links rein" (modulo Rechnung mit der Größe des Arrays).

Queue - Interface

Queue Interface:

```
1: public interface Queue {
2:     void enqueue(Object o);
3:     Object dequeue();
4:     Object head();
5:     boolean isEmpty();
6: }
```


Queue - Implementation

Anmerkung

Array wie bei Stack, zwei Variablen *head* und *tail*, die Arrayindizes speichern. Zu Anfang ist $head = tail = 0$. Eine Variable *MAX* für die Größe des Arrays.

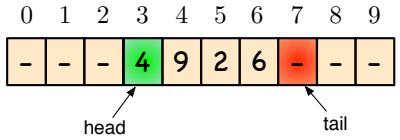
Algorithmus 1 Queue.isEmpty()

```
1: if head == tail then
2:     return true;
3: else
4:     return false;
5: end if
```

Queue - Implementation

Algorithmus 2 Queue.enqueue(Object o)

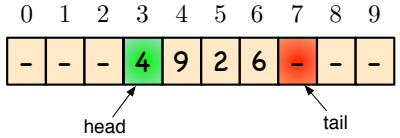
```
1: if ((rear + 1) % MAX) == head then
2:     return ERROR;
3: else
4:     arrQueue[rear] = o;
5:     rear = (rear + 1) % MAX;
6: end if
```



Queue - Implementation

Algorithmus 3 Queue.dequeue()

```
1: if head == rear then
2:     return ERROR;
3: else
4:     Object o = arrQueue[head];
5:     head = (head + 1) % MAX;
6:     return o;
7: end if
```



Zusammenfassung: Queue

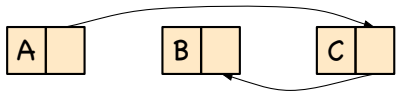
Warteschlange

- Zugriff: FIFO
- Operationen: head, enqueue, dequeue, isEmpty, (size)
- Implementierung:
  - Sequentiell (Array, ggf. zyklisch!)
    - Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Prioritätswarteschlangen

Lineare Liste

- Endliche Folge von Elementen (eines Grundtyps).
- Elemente haben eine Ordnung in der Liste.
- Also: Ähnlich wie bei einem Array.
- ABER: Oft mittels *verketteter Speicherung* implementiert und dann eine Datenstruktur von *dynamischer Größe*!
- Elemente werden in einem *Knoten* abgelegt.
- Knoten hat auch einen "Zeiger" auf das nächste Element in der Liste.

Lineare Liste - Der Knoten




```
1: public class Node {
2:     private Object item;
3:     private Node next;
4: }
```


Lineare Listen - Methoden

Grundlegende Methoden:

- insert(x,p) - füge x an Position p ein
- remove(p) - entferne das Element an Position p
- search(x) - suche Position von Element x
- lookup(p) - Element an Position p
- length() - Länge der Liste
- isEmpty() - ob die Liste leer ist

Hinweis

Man vergleiche obige Operationen mit denen eines Arrays!

Lineare Listen - Speicherung

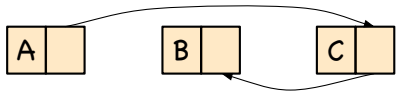
Sequentielle Speicherung (in Arrays) vs. verkettete Speicherung (mit Referenzen/Zeigern)

- Sequentielle Speicherung:
  - \+ schneller Zugriff (auf einzelne Elemente)
  - \- langsames Einfügen/Löschen
- Verkettete Speicherung:
  - \+ schnelles Einfügen/Löschen
  - \- langsamer Zugriff
  - \- (höherer Speicherbedarf)

Lineare Listen - Implementierung

Grundlegende Methodik um eine Liste zu durchwandern:

```
1: p = head;
2: while p != NULL do
3:     tue etwas mit p bzw. p.item
4:     p = p.next
5: end while
```



Lineare Listen - Implementierung

Einfügen von x nach p (und vor $p.next$!)

```
1: Node n = new Node();
2: n.item = x;
3: n.next = p.next;
4: p.next = n;
```

Anmerkung

Zunächst muss man ggf. an das Element p ran. Das macht man mit obigem Durchwandern der Liste!

Lineare Listen - Implementierung

Entfernen von x nach p:

```
1: Node dummy = p.next;
2: p.next = p.next.next;
3: delete(dummy);
```

Hinweis

In Java ist händisches Löschen von dummy i.A. nicht nötig.

Lineare Listen - Wichtige Anmerkungen

Anmerkung

Auf den Listenanfang und das Listenende muss man ganz besonders achten!
Ferner macht man in Java Zugriffe der Art p.next
etc. normalerweise nicht! Hier nutzt man dann entsprechend getter- und setter-Methoden.

Zur Übung

Zur Übung

Die Implementierung einer Datenstruktur *Lineare Listen* (inkl. des Interfaces) ist zur Übung nützlich! Dazu:

- Erst das Interface!
- Dann eine Klasse *LinearList* (oder ähnlich), die eine Referenz auf den Kopf der Liste hat (ein Node) und die Methoden zur Verfügung stellt.
- Eine Klasse *Node*, die eine Referenz auf ein zu speicherndes Objekt enthält und eine Referenz auf einen *Node* - den Nachfolger nämlich.
- Und: Gut dokumentieren!

Doppelt verkettete Listen

Oft ist es hilfreich vor *und* zurück gehen zu können. Dies führt zu *doppelt* verketteten Listen.

```
1: public class Node {
2:     private Object item;
3:     private Node next;
4:     private Node prev;
5: }
```

Neben einen Zeiger auf den Kopf der Liste (head) hat man dann oft auch einen Zeiger auf das Ende der Liste (rear). Die Methoden Einfügen/Löschen werden dann komplizierter....

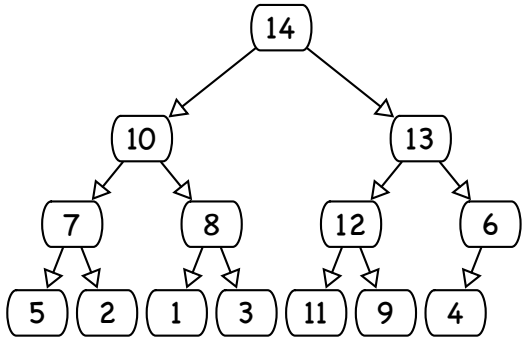
Zusammenfassung: Listen

Listen

- Ähnelt einem Array
- Zugriff: Random (Dauer abhängig von Implementation)
- Operationen: insert(x,p), remove(p), search(x), lookup(p), empty, lenght, (concat, append, head, tail, ...)
- Implementierung:
  - Sequentiell (Array)
    - \+ schneller Zugriff
    - – langsames Einfügen
  - Verkettete Speicherung
    - \+ schnelles Einfügen/Löschen
    - – langsamer Zugriff, (höherer Speicherbedarf)
- *Verkettung* kann *einfach* oder *doppelt* sein; letzteres erlaubt auch Durchlaufen von hinten nach vorne.
- (*Wächter* vereinfachen die Implementierung.)

Bäume

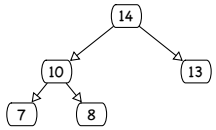
Bäume



Begriffe

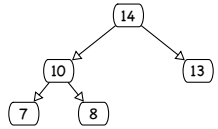
- Knoten: Element eines Baumes
- Vorgänger: direktes vorheriges Element im Baum (Vater, Elternteil)
- Nachfolger: direkt nachfolgendes Element im Baum (Sohn, Kind)
- Vorfahre: Knoten auf dem Weg zur Wurzel
- Nachfahre: Knoten auf dem Weg zu einem Blatt
- Wurzel: Knoten ohne Vorgänger
- Blatt: Knoten ohne Nachfolger
- innerer Knoten: Knoten mit Nachfolger (nicht die Blätter)
- Pfad: Folge von Knoten, die mit Kanten verbunden sind

Begriffe



- Knoten: 7, 8, 10, 13, 14
- Vorgänger: 10 von 7 und 8; 14 von 10 und 13
- Nachfolger: 7 und 8 von 10; 10 und 13 von 14
- Vorfahre: 10 und 14 von 8; 10 und 14 von 7; 14 von 10; 14 von 13
- Nachfahre: andersherum, z.B. 7 von 10, 7 von 14, ...

Begriffe



- Wurzel: 14
- Blatt: 7, 8, 13
- innerer Knoten: 14, 10
- Pfad: (14,10,7) ist ein Pfad von 14 zu 7

Weitere Begriffe

Die bisher dargestellten Bäume waren *Binärbäume* - jeder Knoten hat maximal zwei Kinder. Allgemein kann jeder Knoten eine beliebige (endliche) Anzahl von Knoten haben.

Die Höhe oder Tiefe eines Baumes ist rekursiv definiert:

- Für einen Knoten ohne Kinder (ein Blatt) ist die Tiefe/Höhe 0.
- Für einen inneren Knoten mit Kindern u und v ist die Tiefe/Höhe das Maximum der Tiefe von u und v plus 1.

Alternativ: Maximale Tiefe eines Blattes, wobei die Tiefe eines Knotens k die Anzahl der Kanten von k bis zur Wurzel ist.

Ebene genau andersherum: Wurzel ist Ebene 0, Kinder der Wurzel auf Ebene 1 usw.

Weitere Begriffe

Ein Binärbaum ist *vollständig*, wenn jeder Knoten (bis auf die Blätter) genau zwei Kinder hat.

Für einen vollständigen Binärbaum der Höhe h gilt:

- Anzahl Knoten auf Ebene i: 2^i
- Anzahl Blätter: 2^h (h ist die Höhe)
- Anzahl Knoten: $|T| = \sum_{i=0}^{h} = 2^i = 2^{h+1} - 1$
- Zur Speicherung von $|T|$ Knoten braucht man einen Binärbaum der Höhe $\log_2(|T|+1) - 1 \in \Theta(\log(|T|))$, soll heissen ca. $\log_2(|T|)$ viele.

Bäume - Implementierung

Wie implementiert man Bäume?

So ähnlich wie Listen. Jeder Knoten hat einen Zeiger zum linken und rechten Kind (bei Binärbäumen) und zum Vorgänger.

Zusammenfassung: Bäume

Bäume

- Begriffe: Knoten, Vorgänger, Nachfolger, Wurzel, innerer Knoten, Blatt Höhe, Tiefe, Ebene ($\approx$ Tiefe)
- Speziell: Binärbaum
  - Anzahl Knoten auf Ebene i: 2^i
  - Anzahl Blätter: 2^h (h ist die Höhe)
  - Anzahl Knoten: $|T| = \sum_{i=0}^{h} = 2^i = 2^{h+1} - 1$
  - Zur Speicherung von $|T|$ Knoten braucht man einen Binärbaum der Höhe $\log_2(|T|+1) - 1 \in \Theta(\log(|T|))$, soll heissen ca. $\log_2(|T|)$ viele.

Graphen: Einführung

Graphen sind eine grundlegende Datenstruktur, die in vielen Bereichen der Informatik (und auch in anderen Bereichen) Anwendung findet. Man kann ohne Einschränkung zwei Elemente einer Mengen (den Knoten) in Beziehung setzen (durch eine Kante). Bäume sind spezielle Graphen.

Anmerkung

Erlaubt man verschiedene Kanten-'Typen', so kann man sogar verschiedene Beziehungen ausdrücken.

Definitionen

Definition

Ein *Graph* ist ein Tupel $G = (V, E)$ bestehend aus einer Menge V (auch $V(G)$) von *Knoten* oder Ecken und einer Menge E (auch $E(G)$) von *Kanten*.
Ist G ein *ungerichteter Graph*, so ist

$$E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V, v_1 \neq v_2\},$$

ist G ein *gerichteter Graph*, so ist

$$E \subseteq V^2.$$

Ist $|E|$ viel kleiner als $|V|^2$, so nennt man den Graphen *dünn besetzt*. Ist $|E|$ nahe an $|V|^2$, so spricht man von *dicht besetzten* Graphen.

Definitionen

Definition

- Sind je zwei Knoten von G mit einer Kante verbunden, so ist G ein *vollständiger Graph*. Bei n Knoten: K^n.
- Eine Menge paarweise nicht benachbarter Knoten nennt man *unabhängig*.
- Der *Grad* $d(v)$ eines Knotens v ist die Anzahl mit v inzidenter Kanten.
- Die Menge der *Nachbarn* eines Knotens v bezeichnet man mit $N(v)$ (hier gilt $d(v) = |N(v)|$).
- $\delta(G)$ ist der *Minimalgrad* von G, $\Delta(G)$ der *Maximalgrad*.

Definitionen

Definition

- Ein *Weg* ist ein nicht leerer Graph $P = (V, E)$ mit $V = \{x_0, x_1, \ldots, x_k\}$, $E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$, wobei die x_i paarweise verschieden sind. x_0 und x_k sind die Enden von P, sie sind durch P verbunden. Die Anzahl der Kanten eines Weges ist seine *Länge*.
- Ist P wie oben ein Weg, so ist $P + x_kx_0$ ein Kreis (der Länge $k + 1$).
- Der Abstand zweier Knoten x und y voneinander wird mit $d(x, y)$ bezeichnet und ist die geringste Länge eines x-y-Weges.

Definitionen

Definition

- Seien $G = (V, E)$ und $G' = (V', E')$ Graphen. Gilt $V' \subseteq V$ und $E' \subseteq E$, so nennt man G' einen *Teilgraphen* von G.
- Ist $G = (V, E)$ ein Graph und $V' \subseteq V$, so nennt man den Graphen $G' = (V', E')$ mit $E' = \{\{v_1, v_2\} \in E \mid v_1, v_2 \in V'\}$ den von V' *induzierten Graphen*.

Darstellung von Graphen

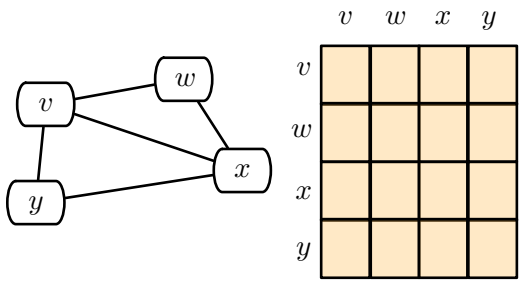
Ein Graph $G = (V, E)$ wird dargestellt indem man seine Knoten als Punkte und die Tupel oder Mengen aus E als (gerichtete) Kanten zwischen die Knoten einzeichnet.

Im Computer speichert man einen Graphen meist mittels einer *Adjazenzmatrix* oder einer *Adjazenzliste*. (Man kann die Mengen V und E aber auch direkt speichern.)

Anmerkung

Bei Graphen schreibt man (und wir) oft $O(V + E)$ etc., wenn $O(|V| + |E|)$ gemeint ist. Man beachtet zudem, dass dies die Komplexität bzgl. der Kenngrößen V und E ausdrückt und nicht umbedingt die Größe der Eingabe wiederspiegelt!

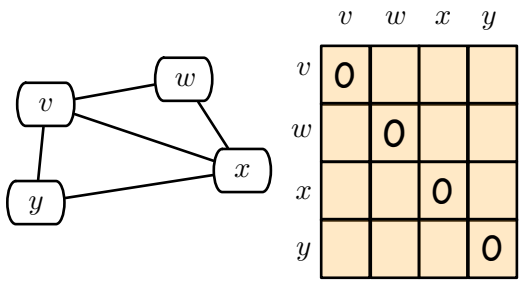
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v				
w				
x				
y				

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

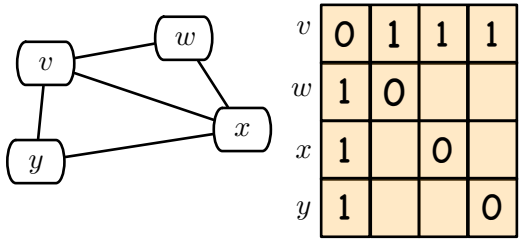
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0			
w		0		
x			0	
y				0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

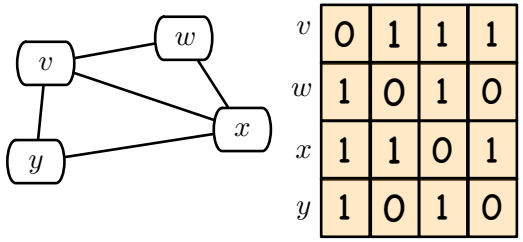
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0	1	1	1
w	1	0		
x	1		0	
y	1			0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

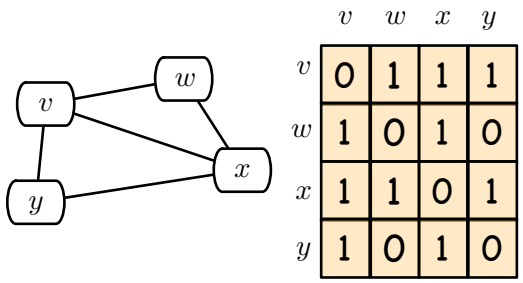
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

Darstellung von Graphen - Adjazenzmatrix

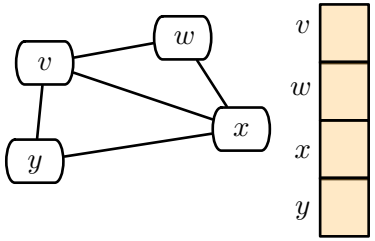


	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

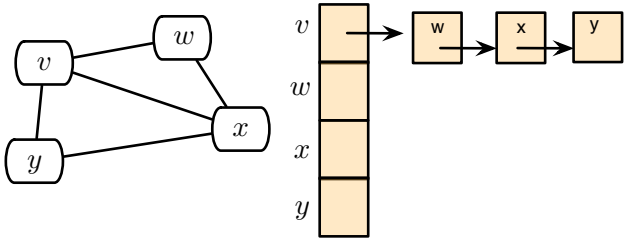
Bei einer Adjazenzmatrix hat man eine $n \times n$-Matrix, bei der an der Stelle (i, j) genau dann eine 1 steht, wenn v_i und v_j verbunden sind.

Der Speicherplatzbedarf ist in $\Theta(V^2)$ (unabhängig von der Kantenzahl).

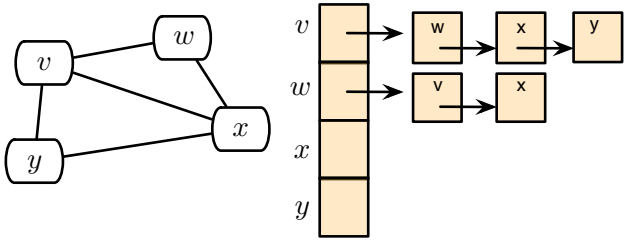
Darstellung von Graphen - Adjazenzlisten



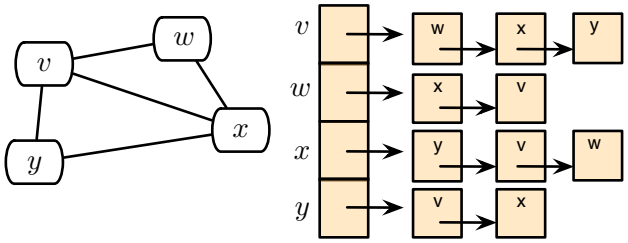
Darstellung von Graphen - Adjazenzlisten



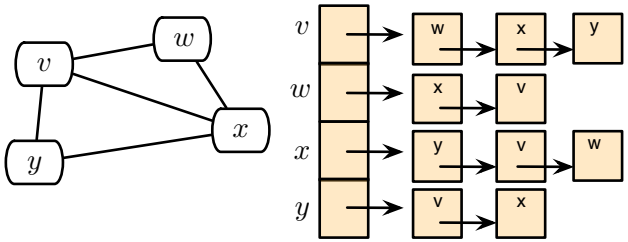
Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Bei der Adjazenzlistendarstellung haben wir ein Array von $|V|$ Listen, für jeden Knoten eine. Die Adjazenzliste $Adj[v]$ zu einem Knoten v enthält alle Knoten, die mit v adjazent sind.

Bei einem gerichteten Graphen ist die Summe aller Adjazenzlisten $|E|$, bei einem ungerichteten Graphen $|2E|$. Der Speicherplatzbedarf ist folglich $\Theta(V + E)$.

Darstellung von Graphen - Zusammenfassung

- Adjazenzmatrix: $|V| \times |V|$-Matrix $A = (a_{ij})$ mit $a_{ij} = 1$ falls $(i, j) \in E$ und 0 sonst. Größe in $\Theta(V^2)$.
- Adjazenzliste: Liste $Adj[v]$ für jeden Knoten $v \in V$ in der die Knoten, die mit v adjazent sind gespeichert sind. Größe in $\Theta(V + E)$.
- Bei einer Adjazenzmatrix kann man schnell herausfinden, ob zwei Knoten benachbart sind oder nicht. Dafür ist es langsamer alle Knoten zu bestimmen, die mit einem Knoten benachbart sind. (Bei Adjazenzlisten genau andersherum.)
- Beide Darstellungen sind ineinander transformierbar.
- Beide Darstellungen sind leicht auf den Fall eines gewichteten Graphen anpassbar.

Zusammenfassung: Stack

Stack

- Zugriff: LIFO
- Operationen: top/head, push, pop, isEmpty, (size)
- Implementierung:
  - Sequentiell (Array)
    - Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Erkennen wohlgeformter Klammerausdrücke
  - Ermitteln der zusammengehoerigen Klammerpaare in $O(n)$ (Idee: Index der öffnenden Klammer auf Stack pushen)
  - Noch schneller geht's ohne Stack indem man einen Zähler bei öffnenden Klammern um eins erhöht und bei schließenden Klammern um eins verringert.

Zusammenfassung: Queue

Warteschlange

- Zugriff: FIFO
- Operationen: head, enqueue, dequeue, isEmpty, (size)
- Implementierung:
  - Sequentiell (Array, ggf. zyklisch!)
    - – Problematisch bei Größenänderungen
  - Verkettete Liste (s.u.)
- Anwendung: Prioritätswarteschlangen

Zusammenfassung: Listen

Listen

- Ähnelt einem Array
- Zugriff: Random (Dauer abhängig von Implementation)
- Operationen: insert(x,p), remove(p), search(x), lookup(p), empty, lenght, (concat, append, head, tail, ...)
- Implementierung:
  - Sequentiell (Array)
    - \+ schneller Zugriff
    - – langsames Einfügen
  - Verkettete Speicherung
    - \+ schnelles Einfügen/Löschen
    - – langsamer Zugriff, (höherer Speicherbedarf)
- *Verkettung* kann *einfach* oder *doppelt* sein; letzteres erlaubt auch Durchlaufen von hinten nach vorne.
- (*Wächter* vereinfachen die Implementierung.)

Zusammenfassung: Bäume

Bäume

- Begriffe: Knoten, Vorgänger, Nachfolger, Wurzel, innerer Knoten, Blatt Höhe, Tiefe, Ebene ($\approx$ Tiefe)
- Speziell: Binärbaum
  - Anzahl Knoten auf Ebene i: 2^i
  - Anzahl Blätter: 2^h (h ist die Höhe)
  - Anzahl Knoten: $|T| = \sum_{i=0}^{h} = 2^i = 2^{h+1} - 1$
  - Zur Speicherung von $|T|$ Knoten braucht man einen Binärbaum der Höhe $\log_2(|T|+1) - 1 \in \Theta(\log(|T|))$, soll heissen ca. $\log_2(|T|)$ viele.

Zusammenfassung: Graphen

Graphen

- Begriffe: Knoten, Kanten, gerichtetet, ungerichtetet, vollständig, unabhängig, Grad, Nachbarn, Weg, Kreis, Abstand, Teilgraph, induzierter Graph, Adjazenzmatrix, Adjazenzliste
- Zu Graphen machen wir später noch viel mehr...