Datenstrukturen und Algorithmen

- Jedes Programm verwendet Datenstrukturen und Algorithmen um seine Aufgabe zu erfüllen
- Diese müssen offenbar zunächst sorgfältig dem speziellen Problem entsprechend ausgewählt werden
- Hat man sich für solche D‘s und A‘s entschieden, müssen diese noch sorgfältig implementiert werden

- Zur Beschreibung und Implementierung der Datenstrukturen und Algorithmen verwenden wir hier C++
- Formulierung von Datenstrukturen als Objekte, Algorithmen als Funktionen
- Saubere Schnittstellenspezifikation und Klassendesign ist mindestens genauso wichtig wie die eigentliche Implementierung
- Der Berufsalltag der meisten Informatiker besteht weit häufiger aus dem Entwickeln solcher Spezifikationen als aus dem Schreiben von Programmcode

Nach welchen Kriterien wählt man Datenstrukturen und Algorithmen für ein spezielles Problem?

- Korrektheit (klar…)
- Laufzeit
- Speicherverbrauch

- Laufzeit und Speicherverbrauch kann man oft mit den selben Techniken analysieren. Daher sprechen wir im Folgenden nur von Laufzeiten.

- Wichtigste Werkzeuge:
  - Asymptotische Notation
  - Rekurrenzgleichungen
  - Worst-Case – und amortisierte Analyse

Asymptotische Notation – Das Wachstum von Funktionen

- Für kleine Problemgrößen spielt die Laufzeit meist keine Rolle. Uns interessiert statt dessen die Laufzeit für *große Probleminstanzen!*
- Daher analysieren wir häufig das Laufzeitverhalten nur für *genügend große* Eingaben (Grenzwertprozess!)
- Dies motiviert die folgenden *Wachstumsklassen:*

  - o(f(n)): alle Funktionen die *„langsamer wachsen“* als f
  - $\mathcal{O}$(f(n)): alle Funktionen die *„nicht schneller wachsen“* als f
  - Θ(f(n)): alle Funktionen die *„so schnell wachsen“* wie f
  - Ω(f(n)): alle Funktionen die *„nicht langsamer wachsen“* als f
  - ω(f(n)): alle Funktionen die *„schneller wachsen“* als f

Genaue Definitionen:

Sei $F = \{ f : N \to R_0^+ \}$ die Menge aller Funktionen von N nach R_0^+.

Definition [1]: Für g∈F ist

$$\Theta(g) = \{ f \in F \mid \exists c_1, c_2 > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

$$O(g) = \{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) \leq cg(n) \}$$

$$\Omega(g) = \{ f \in F \mid \exists c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) \geq cg(n) \}$$

$$o(g) = \{ f \in F \mid \forall c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) < cg(n) \}$$

$$\omega(g) = \{ f \in F \mid \forall c > 0 \wedge \exists n_0 \in N : \forall n \geq n_0 : f(n) > cg(n) \}$$

Häufig lassen sich die folgenden Sätze einfacher anwenden als die Definitionen:

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0 \Rightarrow f(n) \in o(g(n))$$

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = c \text{ mit } 0 < c < \infty \Rightarrow f(n) \in \Theta(g(n))$$

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = \infty \Rightarrow f(n) \in \omega(g(n))$$

Programmierung II

- Damit können wir jetzt Laufzeiten miteinander vergleichen. Aber woher kennen wir die Laufzeit eines Algorithmus?
- Bei *iterativen* Algorithmen ist es oft leicht, einfach die Anweisungen zu *zählen (vgl. z.B. Musterlösung Blatt 4, Aufgabe 4)*
- Die Analyse *rekursiver Algorithmen* führt hingegen meist auf eine sog. *Rekurrenzgleichung*

Beispiel:

```
Mergesort(vector<int> A, int p, int r)   <-- T(n)
{
  if (p<r)
  {
    int q = (p+r)/2;                     <-- Θ(1)
    Mergesort(A,p,q);                    <-- T(n/2)
    Mergesort(A,q+1,r);                  <-- T(n/2)
    Merge(A,p,q,r);                      <-- Θ(n)
  }
}
```

$$\Rightarrow T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)$$

Programmierung II

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)$$

Dies ist eine *Rekurrenzgleichung,* denn T hängt vom Wert von T für kleinere Eingaben ab!

Wie löst man Rekurrenzgleichungen?

Wir haben zwei Verfahren dafür kennengelernt:

- Substitutionsmethode
- Mastertheorem

Lösung mit der Substitutionsmethode:

- „Rate“ eine Lösung (z.B. über den Rekursionsbaum)
- Beweise die Korrektheit dieser Lösung per Induktion

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Programmierung II

Lösung mit dem Mastertheorem:

- Mit dem Mastertheorem kann man sehr einfach Rekurrenzen der Form T(n) = aT(n/b) + f(n) lösen
- Leider hat es aber zwei *Definitionslücken*, d.h. es gibt einige Rekurrenzen dieser Form, die nicht mit dem Mastertheorem lösbar sind
- Idee des Mastertheorems: vergleiche die Kosten für
  - Aufspalten in Teilprobleme / Zusammensetzen zur Gesamtlösung ($n^{log[b](a)}$)
  - Lösen der Teilprobleme (f(n))

Der Vergleich der Kosten Aufspalten / Lösen kann offenbar drei verschiedene Ergebnisse haben:

1. Aufspalten/Zusammensetzen ist teurer
   ⇒ Aufspalten/Zusammensetzen dominiert die Kosten
   ⇒ $\approx \mathcal{O}(n^{\log[b](a)})$
2. Kosten in etwa gleich ⇒ Kosten dominiert durch Aufspalten/Zusammensetzen *und* Lösen
   ⇒ $\approx \mathcal{O}(\log(n)\, n^{\log[b](a)})$
3. Lösen ist teurer ⇒ Lösen dominiert die Kosten
   ⇒ $\approx \mathcal{O}(f(n))$

Kleiner „Haken“: „teurer“ reicht leider nicht aus! Wir müssen bei 1. und 3. statt dessen „wesentlich teurer“ verlangen!

Unter „wesentlich teurer“ verstehen wir, dass zwischen die beiden Funktionen noch mindestens ein Polynom passt, der Unterschied also mindestens *polynomiell* ist!

Wir landen also bei den folgenden drei Fällen ($a \geq 1$, $b > 1$, $f(n) \geq 0$):

$$T(n) = aT(n/b) + f(n)$$

(1) Ist $f(n) = O(n^{\log_b(a)-\varepsilon})$ für ein $\varepsilon > 0$, so gilt: $T(n) = \Theta\left(n^{\log_b(a)}\right)$

Erste Lücke

(2) Ist $f(n) = \Theta(n^{\log_b(a)})$ dann ist: $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$

Zweite Lücke

(3) Ist $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ für ein $\varepsilon > 0$, und ist $a\, f(n/b) \leq c\, f(n)$ für eine Konstante $c < 1$ und genügend großes n, so gilt: $T(n) = \Theta(f(n))$

Je nach Anwendung verwendet man verschiedene Arten von Laufzeiten:

- Worst-Case – Laufzeiten (schlimmer kann‘s nicht werden!)
- Erwartete Laufzeiten (Statistik über die möglichen Eingaben)
- Amortisierte Laufzeiten (in einer Sequenz von Operationen kann häufig nicht jedesmal der worst-case auftreten) bessere Laufzeit der Gesamtsequenz.
  Beispiel: Sequenz von *clear-Operationen* auf einem vector. Das erste clear läuft in O(n), die anderen in O(1), da ja nichts mehr zu tun ist.)

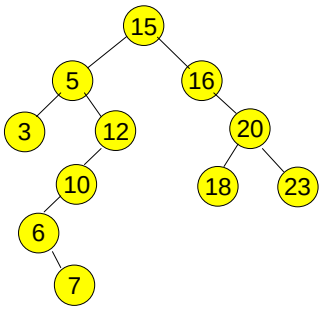
Programmierung II

- Nachdem wir nun Laufzeiten bestimmen und interpretieren können, können wir einen *Grundstock* von allgemein verwendbaren Datenstrukturen und Algorithmen aufbauen
- Die am häufigsten benötigten Operationen einer Datenstruktur sind *einfügen, löschen* und *suchen,* daher sollte man von jeder Datenstruktur wissen, ob und wie effizient sie diese zur Verfügung stellt
- Aus diesen Datenstrukturen kann man dann die dem jeweiligen Problem am besten angepasste auswählen
- Meist ist die Wahl von Datenstrukturen und Algorithmen *wesentlich* wichtiger für die Laufzeit eines Programms als ein hoch optimierter Programmcode
  $\Rightarrow$ ein *„guter Informatiker“* sollte möglichst viele Datenstrukturen mit ihren Vor- und Nachteilen kennen

Datenstrukturen aus der Vorlesung:

- Unbounded arrays (vector in C++)
- Listen (einfach verkettet, doppelt verkettet, zyklisch,…)
- (Such-)Bäume und Wälder (binär, Rot-Schwarz, AVL, …)
- Heaps (Binär, d-när, Binomial, Fibonacci)
- Graphen (gerichtet, ungerichtet)

Binäre Suchbäume

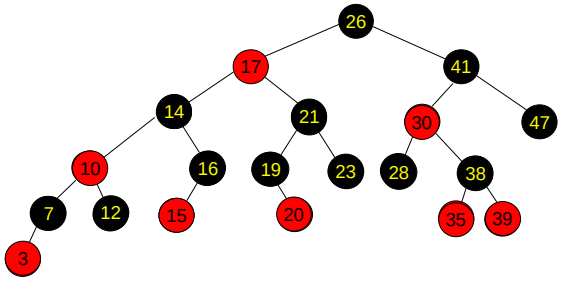


Daten im linken Teilbaum immer $\leq$, im rechten Teilbaum immer $\geq$ den Daten in der Wurzel

Einfügen/Löschen/Suchen: worst-case in $\mathcal{O}$(h) (Höhe des Baumes)

$\Rightarrow$ Wir wollen möglichst geringe Höhe schon beim Aufbau des Baumes garantieren

⇒ Rot-Schwarz - Bäume

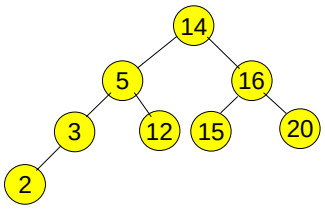


Die Bedingungen an die Farbe eines Knotens und damit an die Baumstruktur garantieren, dass kein Pfad von der Wurzel zu den Blättern mehr als doppelt so lang sein kann als ein anderer solcher Pfad

⇒ der Baum ist beinahe balanciert

⇒ Einfügen/Löschen/ Suchen in $\mathcal{O}$(log(n))

AVL-Bäume

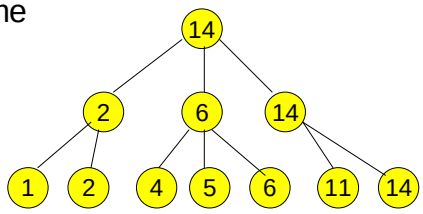


Baum ist soweit wie möglich balanciert

(die Anzahl Knoten in jedem linken und jedem rechten Teilbaum unterscheidet sich um maximal 1)

⇒ Einfügen/Löschen/Suchen in $\mathcal{O}$(log(n))

2-5 - Bäume



Balanciert und simpel, verschenken aber viel Speicher

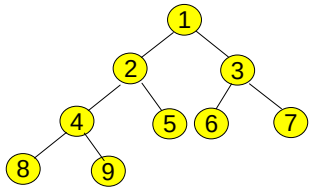
Heaps

Ein Heap ist ein Suchbaum, der die *Heapeigenschaft* erfüllt:

Die Kinder sind immer extremer als die Eltern!

Min-Heap: Wert der Eltern kleiner als der der Kinder

Max-Heap: Wert der Eltern kleiner als der der Kinder



Heaps

- Einfügen geht bei Heaps effizient ($\mathcal{O}$(log(n))
- Effizient *Löschen* können wir aber nur das maximale Element (max-Heap) bzw. das minimale Element (min-Heap)
- Auch die *Suche* ist nicht wirklich schnell

Wofür *braucht* man dann denn überhaupt Heaps?

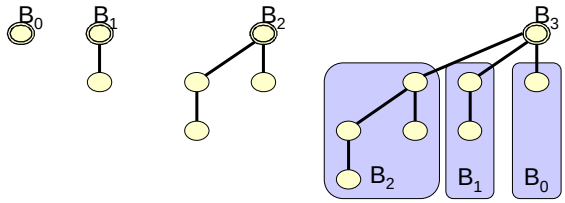
- Heaps können gut sortieren (Heapsort)
- Heaps ergeben gute *Priority Queues* („intelligente ToDo-Liste“ mit den Operationen insert, maximum, extract_max)

Heaps

- Dummerweise will man oft zwei Priority Queues zu einer zusammenfassen
- Das mag unser binärer Heap allerdings gar nicht ($\Theta(n)$)

$\Rightarrow$ Binomialheap

Ein Binomialheap ist ein Wald von *Binomialbäumen,* die die Heapeigenschaft erfüllen, wobei jeder Baumgrad im Wald nur einmal vorkommt



- Binomialheaps kann man in $\mathcal{O}(\log(n))$ vereinigen! ☺
- Leider braucht man häufig noch eine weitere Operation… ☹
- Z.B. speichern viele Graphalgorithmen Kantengewichte in einer Art Priority Queue, wobei jedes Kantengewicht beim besuchen der Kante um 1 dekrementiert wird

  $\Rightarrow$ Wir benötigen ein decrease_key
- Leider hat decrease_key auf Binomialheaps

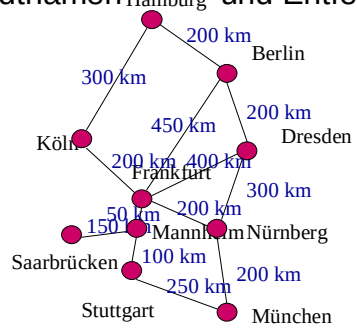
  eine Laufzeit von $\Theta(\log(n))$. Wenn wir das für jede Kante eines dichten Graphen aufrufen, haben wir Laufzeit $\mathcal{O}(\log(|E|) * |E|) = \mathcal{O}(\log(|V|) * |V|^2)$

⇒Fibonacci-Heaps

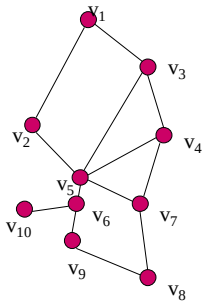
- Fibonacci-Heaps sind Verwandte der Binomialheaps, unterstützen aber insert, merge und decrease_key in amortisiert *konstanter Zeit!*
- ⇒ eine für die Theorie sehr schöne Datenstruktur (z.B. verwendet von den asymptotisch schnellsten MST- und SSSP – Algorithmen)
- Aber:
  - Hohe Konstanten in der O-Notation versteckt!
  - Implementierung und Funktionsweise gruselig!

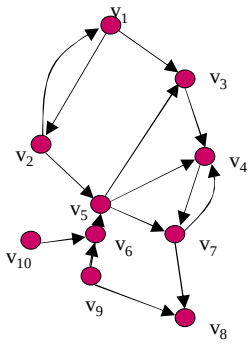
Graphen

- Vielleicht die wichtigste und allgemeinste Datenstruktur der Informatik
- Graphen verbinden Knoten über Kanten miteinander
- Knoten und Kanten können jeweils Informationen tragen (z.B. Stadtnamen und Entfernungen)

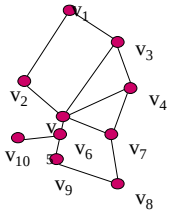


Graphen können *ungerichtet* oder *gerichtet* sein!

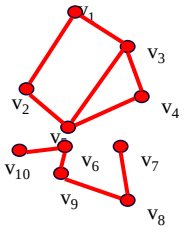




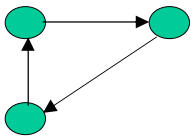
Ein ungerichteter Graph heißt *zusammenhängend (connected)*, wenn man von jedem Knoten aus jeden anderen Knoten über einen Pfad erreichen kann.



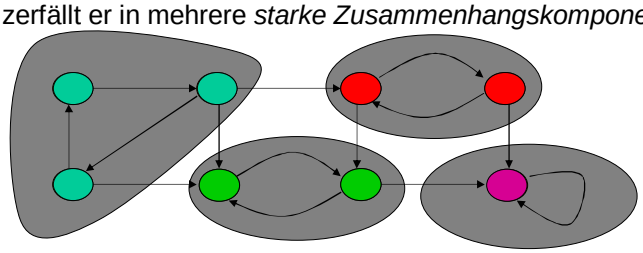
Sonst zerfällt er in mehrere *Zusammenhangskomponenten.*



Ein gerichteter Graph heißt *stark zusammenhängend (strongly connected)*, wenn man für jedes Knotenpaar u und v u von v aus und v von u aus erreichen kann.



Sonst zerfällt er in mehrere *starke Zusammenhangskomponenten.*

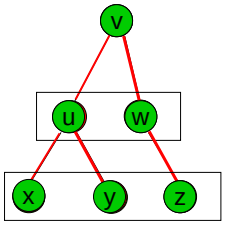


Durchsuchen von Graphen

- Sehr häufig will man einen Graphen nach einem Element mit bestimmten Eigenschaften durchsuchen
- Bsp: In der Künstlichen Intelligenz werden oft die Bewertungen möglicher Züge in einem Spiel in Graphen gespeichert. Der Graph wird dann nach dem Zug mit der höchsten Bewertung durchsucht
  (z.B. bei Schach, Dame, Tic Tac Toe,…)
- Die *Reihenfolge,* in der die Knoten besucht werden, ist häufig von Bedeutung (Bsp: Spielbaum beim Schach: Tiefensuche folgt zuerst *einer* möglichen Zugkombination bis ans Ende des Spiels…)

Breitensuche:

ausgehend von einem Knoten v, durchsuche die von ihm erreichbaren Knoten in der Reihenfolge Ihrer Distanz von v



1.

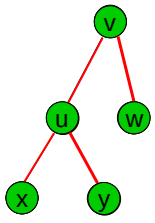
2.

3.

Für den Spezialfall eines Baumes gilt also: der Baum wird Ebene für Ebene durchsucht!

Tiefensuche:

Steige immer so tief wie möglich in den Baum hinab, d.h. folge immer den Kanten des letzten „neu entdeckten“ Knoten.



- Beide Suchverfahren laufen in linearer Zeit in der Anzahl der Kanten und Knoten: $\mathcal{O}(|V| + |E|)$
- Beide Suchverfahren liefern eine zusätzliche Information: den sogenannten *Predecessor-Graphen*
- Im Predecessor-Graphen gibt es eine Kante von u nach v, genau dann wenn wir bei der Suche den Knoten v vom Knoten u aus kommend entdeckt haben!
- Dieser Graph ist eine wichtige Eingabe für viele darauf aufbauende Graphalgorithmen! Daher ist es häufig nützlich sich daran zu erinnern, welche Informationen er enthält!

Beispiele für wichtige Graphalgorithmen:

- Bestimmen der starken Zusammenhangskomponenten (basiert auf DFS)
- Topologisches Sortieren (geht auch mittels DFS)
- Bestimmung des *Minimum Spanning Tree;* dies ist ein Baum, der alle Knoten eines Graphen verbindet, und für den die Summe der Kantengewichte (geklaut aus dem Originalgraphen) minimal ist

  Wichtiges Beispiel für einen Greedy-Algorithmus! (läuft immer in Richtung des aktuellen lokalen Optimums)
- Bestimmen von *kürzesten Pfaden,* d.h. Pfaden mit minimalen Kantengewichten (SSSP, SDSP usw.)