Kapitel 1: Grundlagen

- Analyse von Algorithmen
  - Worst Case Analyse
  - Average Case Analyse
  - Amortisierte Analyse: erlaubt Worst Case Aussagen für Sequenzen von Operationen
- Kostenmaß
  - Einheitskostenmaß: Anzahl Operationen *ohne* Berücksichtigung der Bitlänge der beteiligten Zahlen
  - logarithmisches Kostenmaß: Anzahl Bitoperationen

Kapitel 2: Höhere Datenstrukturen

1. *”Suchbäume”*
   - Operationen: *Find, Insert, Delete, FindMin, FindMax, FindPred, FindSucc*
   - AVL-Bäume, (a, b)-Bäume, Splay Trees
2. *”Vorrangwarteschlange” (”priority queue”)*
   - Operationen: *Insert, Delete, ExtractMin, DecreaseKey*
   - Fibonacci Heaps, k-level Buckets, Radix Heaps
3. *”Mengen”*
   - Operationen: *Find, Union*
   - Union-Find Strukturen
4. *”Wörterbuch” (”dictionary”)*
   - Operationen: *Find, Insert, Delete*
   - Hash–Tabellen, Suchbäume

(a, b)-Bäume, AVL-Bäume

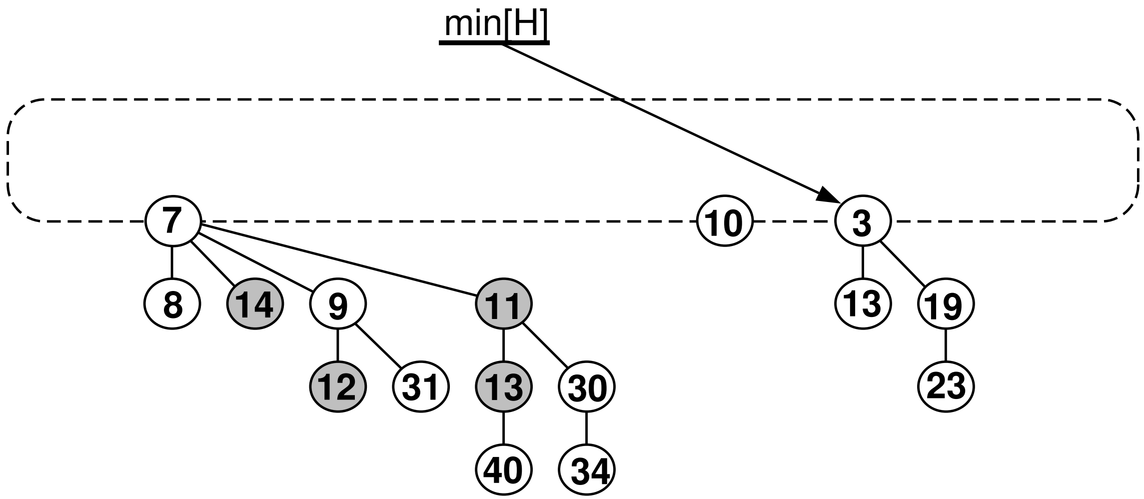
- höhenbalancierte Bäume:
  $\longrightarrow$ maximale Höhe beschränkt durch $1 + \log_a n$ bzw. $\lceil \log_2 n \rceil$
- *Find, Insert, Delete, FindMin, FindMax, FindPred, FindSucc* jeweils $O(\log n)$

Splay Trees

- binäre, nicht notwendigerweise höhenbalancierte Bäume
- Idee: sorge dafür, dass Elemente, auf die „vor kurzem“ zugeriffen wurde, in „der Nähe“ der Wurzel sind
- amortisierte Analyse zeigt: ausgehend von einer leeren Datenstruktur benötigen k beliebige Operationen *Find, Insert, Delete, FindMin, FindMax* höchstens Zeit $O(k \cdot \log n)$

Fibonacci Heaps

- Collection of ordered trees:
  the key of each node is less than or equal to the keys of all its children.
- The roots of these trees are contained in a (doubly linked) cyclic list (*root list*).
- At any time the pointer $\min[H]$ points to the node with the smallest key in the root list (and thus to the node with the smallest key in the entire heap).



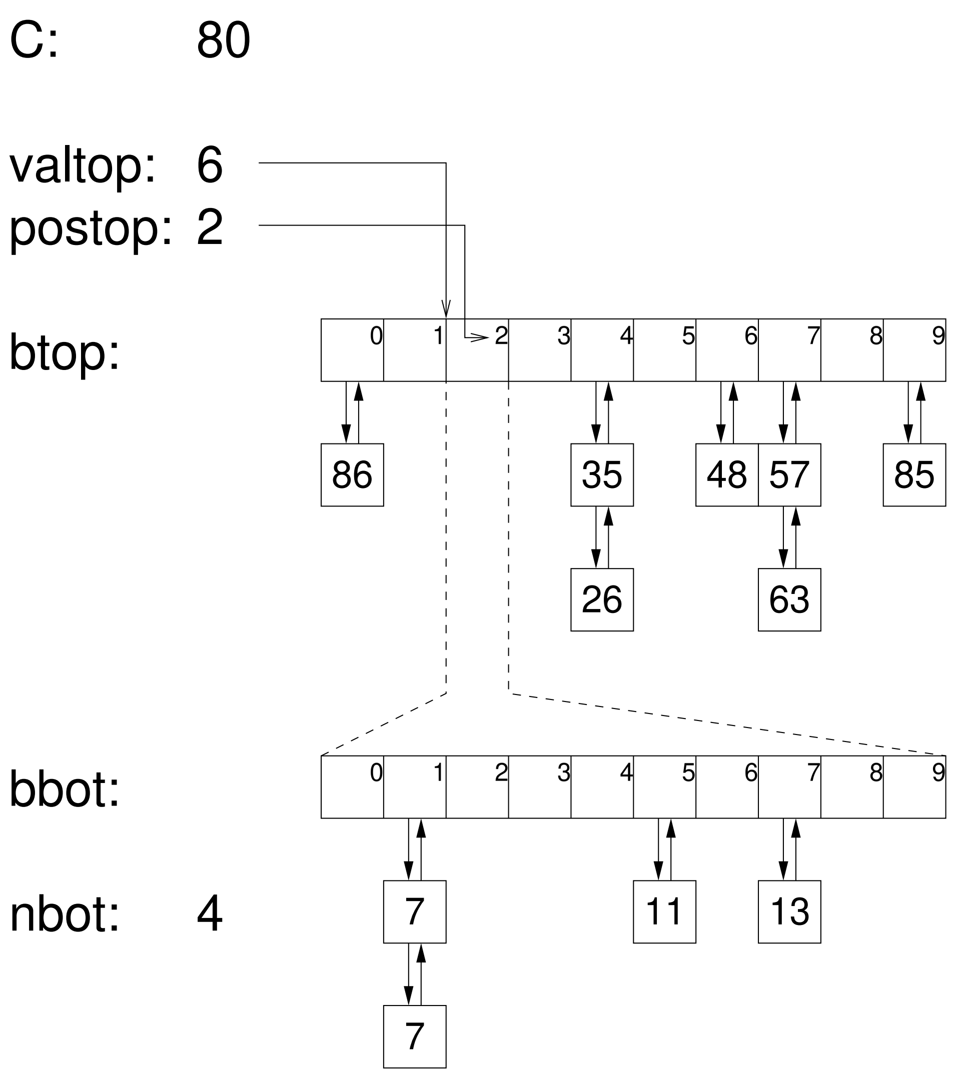
2-level Buckets

Idee: 2-level Buckets verwenden zwei Arrays der Länge jeweils $B = \lceil \sqrt{C+1} \rceil + 1$.

Voraussetzungen an die Schlüssel:

- Die Schlüssel sind nicht-negative ganze Zahlen.
- Zu jedem Zeitpunkt gilt:

  maximaler Schlüssel - minimaler Schlüssel $\leq C$.
- Wenn durch `extractMin` ein Element mit Schlüssel k ausgegeben wurde, so werden danach nur noch Elemente mit Schlüssel $\geq k$ eingefügt. *(Monotonie des Heaps)*



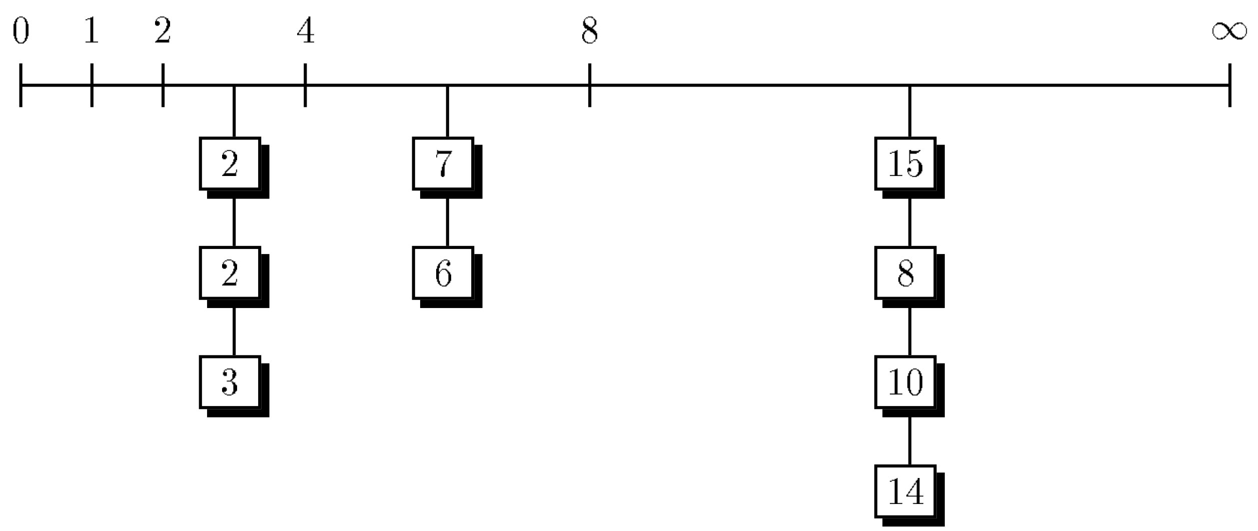
Radix Heaps

Ziel: Reduziere amortisierte Laufzeit der langsamsten Zugriffsfunktion von $\mathcal{O}(\sqrt{C})$ bei 2–level–Buckets auf $\mathcal{O}(\log C)$.

Voraussetzungen an die Schlüssel:

- Die Schlüssel sind nicht-negative ganze Zahlen.
- Zu jedem Zeitpunkt gilt:

  maximaler Schlüssel - minimaler Schlüssel $\leq C$.
- Wenn durch `extractMin` ein Element mit Schlüssel k ausgegeben wurde, so werden danach nur noch Elemente mit Schlüssel $\geq k$ eingefügt. *(Monotonie des Heaps)*



Beispiel eines gefüllten Heaps

Priority Queues – Anwendungen

Zwei Standard-Anwendungen sind:

1. Das kürzeste Wege Problem in einem Graphen: Algorithmus von Dijkstra

Mit Fibonacci Heaps: $\mathcal{O}(m + n \log n)$

Falls Kantengewichte nicht-negative ganze Zahlen und C das größte Gewicht einer Kante:

Mit 2-level Buckets: $\mathcal{O}(m + n\sqrt{C})$

Mit Radix Heaps: $\mathcal{O}(m + n \log C)$

2. Minimal spannenden Bäume in einem Graphen: Algorithmus von Prim

Mit Fibonacci Heaps: $\mathcal{O}(m + n \log n)$

Ergebnis der Programmieraufgabe „Implementiere Dijkstra“

abgegebene Programme: 36

Graph 1: $n = 10, m = 20$

korrektes Ergebnis: 33

Graph 2: $n = 1000, m = 4000$

korrektes Ergebnis und Laufzeit ≤ 1 min: 15

Graph 3: $n = 100.000, m = 400.000$

korrektes Ergebnis und Laufzeit ≤ 20 min: 5

„korrekte“ Implementierung: 2 (Laufzeit jeweils ≤ 1 sec.)

schnellste Lösung: 0.6 sec

Mengen / Union-Find-Strukturen

Gegeben: Datensätze, partitioniert in (disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Ziel: Datenstruktur, die die folgenden drei Operationen möglichst effizient realisiert:

`MakeNewSet(`x`)`	$\rightarrow$	Füge neuen Datensatz x ein; dieser bildet eine neue (einelementige) Menge
`Find(`x`)`	$\rightarrow$	Gebe Repräsentanten der Menge aus, zu der x gehört
`Union(`x`, `y`)`	$\rightarrow$	Vereinige die beiden Mengen mit Repräsentanten x und y, wobei x Repräsentant der neuen Menge sei.

Realisierung: Jede Menge wird durch einen zur Wurzel hin gerichteten Baum (sog. intree) dargestellt, an der Wurzel notieren wir den Repräsentanten der Menge.

Grundversion

`MakeNewSel(`x`):` mache x zur Wurzel eines neuen (einelementigen) Baumes

`Find(`x`):` Laufe von x zur Wurzel r, gebe Repräsentanten von r aus

`Union(`x`, `y`):` Hänge Wurzel des kleineren der beiden Bäume ("kleiner" $\hat{=}$ gesamte Anzahl der Knoten im Baum) an die Wurzel des größeren an.

Lemma: Für jeden in der Union-Find-Struktur enthaltenen Baum T gilt

$$\text{size}(T) \geq 2^{\text{height}(T)}$$

Path Compression

Idee: Ändere `Find`(x) so ab, daß alle Knoten auf dem Pfad von x zur Wurzel r *direkt* unter die Wurzel gehängt werden. (Dadurch erhöht sich die Laufzeit von `Find` nur um einen konstanten Faktor!)

Satz: Ausgehend von einer leeren Datenstruktur benötigen k beliebige Operationen, darunter genau n `MakeNewSet`, höchstens $\mathcal{O}(k \log^* n)$ Schritte.

Union-Find-Strukturen: Anwendungsbeispiel

Eine Standard-Anwendungen ist:

<u>`Kruskals Algorithmus`</u>

Eingabe: zshgd. Graph $G = (V, E)$; Gewichte $w[e]$;

Ausgabe: Kantenmenge T eines min. spann. Baums in G.

Sortiere die Kantenmenge so, daß gilt:

  $w[e_1] \leq w[e_2] \leq \ldots \leq w[e_m]$;

$T := \emptyset$;

for all $v \in V$ **do** MakeNewSet(v);

for i := 1 **to** m **do**

  Bestimme $u, v \in V$ mit $e_i = \{u, v\}$;

  if Find(u) $\neq$ Find(v) **then**

    $T := T \cup \{e_i\}$;

    Union(Find(u), Find(v));

Laufzeit: $\mathcal{O}(m \log n + m \log^* n)$, wobei der erste Term entfällt, wenn die Kanten bezüglich ihrer Gewichte vorsortiert sind. - Durch eine geeignete Modifikation kann man diese Kosten auf $O(m + n \log n)$ reduzieren (vgl. ÜA 2, Blatt 5).

Kapitel 3: Sortieren und Selektieren

- Analyse von QuickSort
- RadixSort
- Selektionsalgorithmus nach BFPRT
- Randomisierter Selektionsalgorithmus
- Untere Schranke für Median

QuickSort

Satz: Quicksort benötigt zum Sortieren eines Feldes der Länge n maximal $\binom{n}{2}$ viele Vergleiche.

Satz: Quicksort benötigt zum Sortieren eines Feldes der Länge n *durchschnittlich* nur $2\,n\ln(n) + O(n)$ viele Vergleiche.
Beweis: Z.B. mit Hilfe rekursiver Funktionen, vgl DS1-Buch.

Random-QuickSort

Idee: Wähle in jedem Schritt das Pivot-Element *zufällig*.

Satz: Random-Quicksort benötigt zum Sortieren eines Feldes der Länge n *im Erwartungswert* nur $2\,n\ln(n) + O(n)$ viele Vergleiche — egal wie die Elemente zu Beginn sortiert sind.

Forward Radix Sort mit Gruppen

$\longrightarrow$ Andersson, Nilsson 1994

Laufzeit für n Strings der Länge k über einem Alphabet Σ, $|\Sigma| = t$: $\mathcal{O}(n + t \cdot k')$, wobei k' kleinste Zahl ist, so dass alle Prefixe der Länge k' verschieden sind.

Beispiel: 179, 208, 306, 093, 859, 055, 009, 271, 033, 984

(1) Sortiere nach 1. Stelle:

0	1	2	3	4	5	6	7	8	9
093;1	179;1	208;1	306;1					859;1	984;1
055;1		271;1							
009;1									
033;1									

$$\Longrightarrow \quad \underbrace{093, 055, 009, 033}_{1}, \underbrace{179}_{5}, \underbrace{208, 271}_{6}, \underbrace{306}_{8}, \underbrace{859}_{9}, \underbrace{984}_{10}$$

(2) Sortiere nach 2. Stelle:

0	1	2	3	4	5	6	7	8	9
009;1			033;1		055;1		271;6		093;1
208;6									

Gr.:	1	2	3	4	5	6	7	8	9	10
	009				179	208		306	859	984
		033					271			
			055							
				093						

Der Algorithmus von Blum, Floyd, Pratt, Rivest, Tarjan

Idee: Teile Elemente in 5er Gruppen auf, bestimme jeweils deren Median; dann bestimme man den Median der Mediane und verwede diesen um mit linearem Aufwand die möglichen Kandidaten um einen linearen Faktor zu reduzieren. Wende Algorithmus rekursiv auf die verbleibenden Elemente an.

Laufzeit: $\mathcal{O}(n)$, $24n$ Vergleiche

Untere Schranke

Dor, Zwick 1996: Es gibt ein $\epsilon > 0$, so dass jeder vergleichbasierte Algorithmus zur Bestimmung des Medians einer n-elementigen Menge mindestens $(2+\epsilon)n - o(n)$ Vergleiche benötigt.

Vorlesung: Jeder vergleichbasierte Algorithmus zur Bestimmung des Medians einer n-elementigen Menge benötigt mindestens $\frac{3}{2}n - 2$ Vergleiche.

Beweisidee: Gegenspieler-Argument

RandomizedMedian

<u>**Idee:**</u> Wähle $n^{3/4}$ Elemente zufällig aus, sortiere diese und wähle an Hand der sortierten Liste zuverlässige „Schätzer“ für den Median. Bestimme in linearer Zeit $n^{3/4}$ möglichen Kandidaten für den Median; bestimme den Median in dem diese Kandidaten sortiert werden.

<u>**Satz**</u>: Der Algorithmus `RandomizedMedian` benötigt zur Bestimmung des Medians einer n-elementigen Menge mit hoher Wahrscheinlichkeit nur $\frac{3}{2}n + o(n)$ viele Vergleiche.

Kapitel 4: Graphalgorithmen

- Das kürzeste Wege Problem
- Topologische Sortierung
- Starker Zusammenhang
- Blockzerlegung

Der Algorithmus von Floyd & Warshall

Idee: Dynamische Programmierung

$F_k[i, j] :=$ Länge eines kürzesten i-j-Pfades an, der als **innere Knoten** (das sind die Knoten des Pfades außer den Endknoten i und j) nur Knoten aus $\{1, \ \ldots, \ k\}$ verwendet.

Der Algorithmus von Bellman & Ford

Idee: Dynamische Programmierung

$B_k[i] :=$ Länge eines kürzesten s-i-Pfades, der aus höchstens k Kanten besteht.

Bem: Beide Algorithmen lassen sich so modifizieren, dass sie auch bei negativen Kantengewichten funktionieren (d.h. negative Kreise entdecken, falls solche existieren).

Der Algorithmus von Johnson

Ziel: Lösung des “all pairs”–Problem mit einer beliebigen (negativen) Längenfunktion ℓ

Idee: Verwende eine so genannte *Rekalibrierung*:

$f\,:\,\mathrm{V}\;\mapsto\;\mathbb{R}$ heißt eine **Rekalibrierung** eines gerichteten Graphen $D = (V; A)$ mit Gewichtsfunktion $\ell\,:\,\mathrm{V}\;\mapsto\;\mathbb{R}$, falls $\forall\,(u, v)\;\in\;\mathrm{A} : f(u) + \ell(u, v) \geq f(v)$ gilt.

Ansatz für Algorithmus: Bestimme zuerst eine Rekalibrierung f und löse dann das “all pair shortest path”–Problem bzgl. ℓ_f (z.B. mit Dijkstra).

Bestimmung der Rekalibrierung: Füge zu dem Graphen einen zusätzlichen Knoten ξ hinzu und verbinde ξ mit jedem anderen Knoten durch eine Kante der Länge 0 verbindet. Verwende nun den Algorithmus von Bellman & Ford, um in dem neuen Graphen kürzeste ξ-v Pfade zu bestimmen, für alle $v \in V$.

Das Kürzeste-Wege-Problem

	$\ell \geq 0$	ℓ beliebig
Startknoten s	Dijkstra: $\mathcal{O}(m + n \log n)$	Bellman-Ford: $\mathcal{O}(nm)$
Alle Paare	Dijkstra: $\mathcal{O}(nm + n^2 \log n)$ Floyd-Warshall: $\mathcal{O}(n^3)$	Bellman-Ford & Dijkstra: $\mathcal{O}(nm + n^2 \log n)$ Floyd-Warshall: $\mathcal{O}(n^3)$

Topologische Sortierung

Definition:

Sei $D = (V, A)$ ein gerichteter Graph. Eine **topologische Sortierung** von D ist eine bijektive Abbildung $\pi : V \to \{1, \ldots, |V|\}$, so dass

$$\pi(u) < \pi(v) \qquad \forall (u, v) \in A.$$

Satz:

Für einen gerichteten Graphen $D = (V, A)$ existiert genau dann eine topologische Sortierung, wenn er azyklisch ist.

Für einen azyklischen gerichteten Graphen $D = (V, A)$ kann man eine topologische Sortierung in $\mathcal{O}(|V| + |A|)$ durch eine modifizierte Tiefensuche bestimmen.

Stark zusammenhängende Graphen

Definition Ein gerichteter Graph $D = (V, A)$ heißt ***stark zusammenhängend***, falls es für alle $u, v \in V$ einen gerichteten Pfad von u nach v gibt.

Satz:

Für einen gerichteten Graphen $D = (V, A)$ kann man durch eine modifizierte Tiefensuche in $\mathcal{O}(|V| + |A|)$ testen, ob er stark zusammenhängend ist,

Artikulationsknoten, Blockzerlegung

Definition Sei G ein zusammenhängender Graph. Ein Knoten $v \in V$ heißt ***Artikulationsknoten*** (oder ***trennender Knoten***), falls $G - v$ nicht zusammenhängend ist.

Ein ***Block*** in einem Graphen G ist ein maximaler zusammenhängender Subgraph von G auf mindestens zwei Knoten, der keinen trennenden Knoten enthält.

Satz:

Für einen Graphen $G = (V, E)$ kann man eine Blockzerlegung in $\mathcal{O}(|V| + |E|)$ durch eine modifizierte Tiefensuche bestimmen.

Kapitel 5: Ausgewählte Verfahren

- Matrizenmultiplikation nach Strassen

  $\longrightarrow$ besser als $\mathcal{O}(n^3)$ ...

- String Matching

  $\longrightarrow$ einfache „Tricks“ um das sture „Durchprobieren“ zu beschleunigen

- Suffix Trees

  $\longrightarrow$ Aufbau einer effizienten Datenstruktur um alle Suffixe eines Wortes abzuspeichern

- Schnelle Fouriertransformation

  $\longrightarrow$ z.B. um Polynome effizient zu multiplizieren

- Huffman Codes

  $\longrightarrow$ Grundideen der Codierungstheorie

Kapitel 6: Approximationsalgorithmen

- Was versteht man unter einem Optimierungsproblem ?

  $\longrightarrow \quad \langle \mathcal{I}, Sol, val, \text{goal} \rangle$

- Wie misst man die Güte eines Approximationsalgorithmus ?

  $\longrightarrow$ relative Güte:

$$\frac{1}{\rho} \;\leq\; \frac{\mathcal{A}(I)}{\textsc{Opt}(I)} \;\leq\; \rho \qquad \text{für alle } I \in \mathcal{I}.$$

- Definition verschiedener Klassen von Optimierungsproblemen:

$$\mathcal{PO} \subseteq \mathcal{FPTAS} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX} \subseteq \mathcal{NPO}.$$

Falls $\mathcal{P} \neq \mathcal{NP}$, dann

- $\mathcal{FPTAS} \setminus \mathcal{PO} \neq \emptyset$:

  $\longrightarrow$ enthält z.B. $\textsc{Knapsack}$

- $\mathcal{PTAS} \setminus \mathcal{FPTAS} \neq \emptyset$:

  $\longrightarrow$ enthält z.B. $\textsc{Maschinen Scheduling}$

- $\mathcal{APX} \setminus \mathcal{PTAS} \neq \emptyset$:

  $\longrightarrow$ enthält z.B. $\textsc{Max3Sat}$

- $\mathcal{NPO} \setminus \mathcal{APX} \neq \emptyset$:

  $\longrightarrow$ enthält z.B. $\textsc{Travelling Salesman}$