Einführung in die Algorithmik

Inhalt

AVL-Bäume

Balancefaktor	Rotation	Kind
Positiv	Rechts	Links
Negativ	Links	Rechts

Aufpassen(!):

- Falls unter einem positiven Balancefaktor ein negativer steht oder andersherum:
  - ⇨ Tauschen!!!

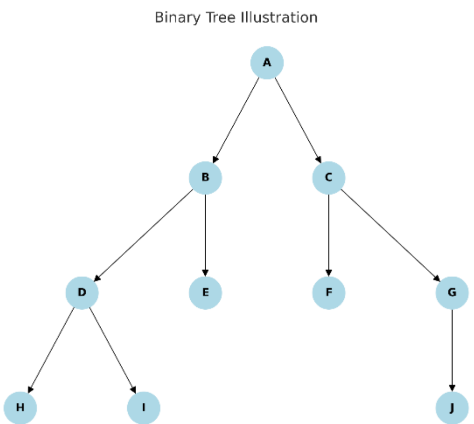
Balancefaktor:

- Höhe-links – Höhe-rechts

Laufzeiten:

- Suche, Einfügen, Löschen: *O(log(n))*

Baumtraverse



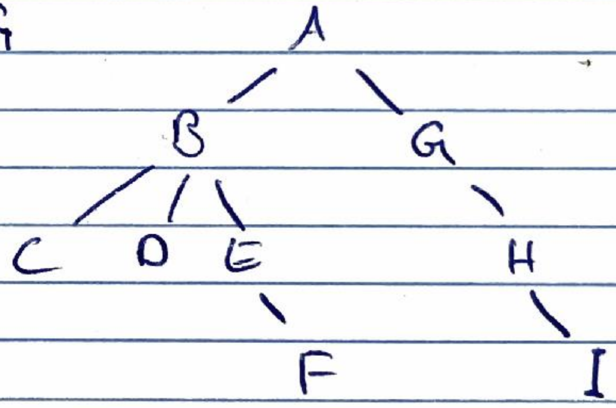
In-Order: HDIBEAFCJG

Pre-Order: ABDHIECFGJ

Post-Order: HIDEBFJGCA

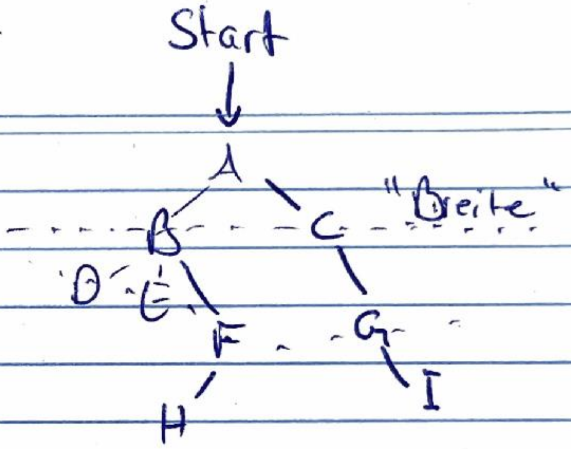
Tiefensuche

	visited	Stack
Stack (LiFo)	A	BG
	AB	CDEG
	ABCD	EG
	ABCDE	FG
	ABCDEF	G
	ABCDEFG	H
	ABCDEFGHI	I



Breitensuche

	visited	Queue
Queue (FiFo)	A	BC
	AB	CDEF
	ABC	DEFG
	ABCD	EFG
	ABCDE	FG
	ABCDEF	G
	ABCDEFG	∅



Laufzeit

Defintion: $\mathcal{O}(f)$

Sei $f : \mathbb{N} \to \mathbb{R}$ eine Funktion.

Die **Klasse** $\mathcal{O}(f)$ **oder** $O(f)$ ist die Menge aller Funktionen $g : \mathbb{N} \to \mathbb{R}$, für die eine reelle Konstante $C > 0$ und eine natürliche Zahl n_0 existieren, so dass

$$|g(n)| \leq C \cdot |f(n)| \quad \text{für alle } n \geq n_0.$$

Etwas formaler

Für $f : \mathbb{N} \to \mathbb{R}$, definieren wir:

$$\mathcal{O}(f) := \left\{ g : \mathbb{N} \to \mathbb{R} \mid \exists C > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : |g(n)| \leq C \cdot |f(n)| \right\}$$

- Konstanten fallen weg
- Multiplikation/Division zweier n-bit langen Zahlen: O(n*n) = O(n)
- Addition: O(n)
- O(f) * O(g) = O(f*g)
- O(f+g) = O(f) -> Absorption

Definition: $\Theta(f)$

Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion.

Die **Klasse** $\Theta(f)$ ist der Schnitt von $\mathcal{O}(f)$ und $\Omega(f)$, d.h.

$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f).$$

Arten von Laufzeit:

Klasse	Sprechweise	Wachstum
$\mathcal{O}(1)$	f ist **beschränkt**	$f(2n) \approx f(n)$
$\mathcal{O}(\log(n))$	f wächst **logarithmisch**	$f(2n) \approx f(n) + C$
$\mathcal{O}(n)$	f wächst **linear**	$f(2n) \approx 2f(n)$
$\mathcal{O}(n\log(n))$	f wächst **super-linear**	$f(2n) \approx 2f(n) + n$
$\mathcal{O}(n^2)$	f wächst **quadratisch**	$f(2n) \approx 4f(n)$
$\mathcal{O}(n^3)$	f wächst **kubisch**	$f(2n) \approx 8f(n)$
$\mathcal{O}(n^k)$	f wächst **polynomiell**	$f(2n) \approx 2^k f(n)$
$\mathcal{O}(2^n)$	f wächst **exponentiell**	$f(n+1) \approx 2f(n)$
$\mathcal{O}(n!)$	f wächst **faktoriell**	$f(n+1) \approx (n+1)f(n)$

Laufzeit Hirachie:

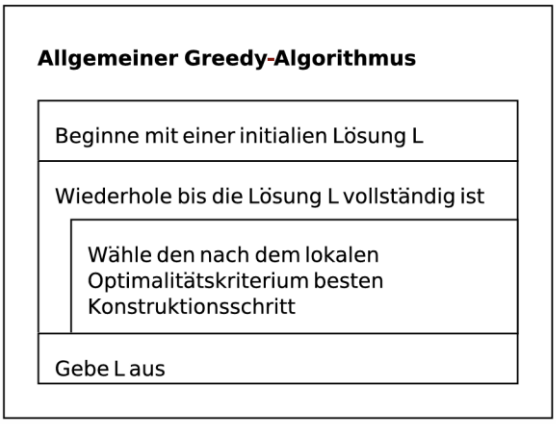
$$\mathcal{O}(1) \subsetneq \mathcal{O}(\log n) \subsetneq \mathcal{O}(n) \subsetneq \mathcal{O}(n \log n) \subsetneq \mathcal{O}(n^2) \subsetneq \mathcal{O}(n^3) \subsetneq \mathcal{O}(n^k) \subsetneq \mathcal{O}(2^n)$$

Sortieren

<table>
<tr><th>Algorithmus</th><th>Worst-Case</th><th>Best-Case</th><th>Average-Case</th><th>Bemerkungen</th></tr>
<tr><td>SelectionSort</td><td rowspan="5">$\mathcal{O}(n^2)$</td><td>$\mathcal{O}(n^2)$</td><td rowspan="5">$\mathcal{O}(n^2)$</td><td></td></tr>
<tr><td>BubbleSort</td><td rowspan="4">$\mathcal{O}(n)$</td><td></td></tr>
<tr><td>InsertionSort</td><td></td></tr>
<tr><td>ShakerSort</td><td></td></tr>
<tr><td>GnomeSort</td><td></td></tr>
<tr><td>HeapSort</td><td rowspan="2">$\mathcal{O}(n \log(n))$</td><td rowspan="3">$\mathcal{O}(n \log(n))$</td><td rowspan="3">$\mathcal{O}(n \log(n))$</td><td></td></tr>
<tr><td>MergeSort</td><td>rekursiv, $\mathcal{O}(n)$ zus. Platz</td></tr>
<tr><td>QuickSort</td><td>$\mathcal{O}(n^2)$</td><td>rekursiv</td></tr>
</table>

Greedy-Algorithmen

- ⇨ Liefern nicht immer die optimale Lösung
- ⇨ Beispiele: Dijkstra, Kruskal, Prim
- ⇨ Voraussetzungen:
    - Teillösungen
    - Grad der Teillösung (höchster Grad = Gesamtlösung)
- ⇨ Kostenfunktion auf der Teillösung



Kruskal-Algorithmus:

<u>Ziel:</u> Ermittlung des minimalen Spannbaums (alle Knoten kostengünstig verbinden) eines zusammenhängenden gewichteten Graphen

<u>Ablauf:</u>

1. Sortierung aufsteigend nach Kantengewicht
2. Auswählen der günstigsten Kante
3. Hinzufügen nächst-günstiger Kanten die keine Kreise bilden
4. Sobald alle Knoten zusammenhängen, ist man fertig

Prim-Algorithmus:

Ziel: Ermittlung des minimalen Spannbaums (alle Knoten kostengünstig verbinden) eines zusammenhängenden gewichteten Graphen

<u>Ablauf:</u>

1. Startknoten auswählen
2. Auswählen der günstigsten Kante der einen neuen Knoten verbindet

Bellman-Ford-Algortihmus:

<u>Ziel</u>: Weg von Knoten A nach Knoten B in einem Graph (kann auch mit negativen Kanten umgehen)

<u>Ablauf</u>:

1. Am Anfang Kosten auf unendlich
2. Tabelle von kürzesten Wegen ZU einem Knoten
3. Check am Ende auf negative Kreise

Dijkstra-Algorithmus:

<u>Ziel:</u> Kürzester Weg bzw. günstigster Weg von Knoten A nach Knoten B in einem zusammenhängenden gewichteten (+) Graphen

<u>Ablauf:</u>

1. Tabelle

Knoten	**A**	**B**	**C**
Kosten	0	Undl.	Undl.
Vorgänger			

2. Warteschlange
   - ⇨ Startknoten, N(A), ...
3. Erledigt
   - ⇨ Nach betrachten von seinen Nachbarn kommt Knoten auf Erledigt
4. Wenn günstigerer Weg gefunden wurde, werden Kosten für jew. Knoten geändert
5. Wenn Warteschlange leer ist, sind wir fertig

Suchbaum Knoten löschen

1. Knoten ist ein Blatt?
   - ⇨ Löschen
2. Knoten besitzt genau ein Kind?
   - ⇨ Knoten löschen und Kind nachrutschen lassen
3. Knoten besitzt 2 Kinder?
   - ⇨ Vorgänger oder Nachfolger identifizieren (jeweils im linken oder rechten Teilbaum ganz rechts oder links) und dieses Blatt als neue Wurzel setzen

Huffman-Code

1. Zählen, wie oft die einzelnen Zeichen im Wort vorkommen
2. Verknüpfe die beiden Zeichen, die am seltensten im Wort vorkommen und schreibe an ihre Wurzel die Summe ihrer Vorkommen
3. Wiederhole 2. So lange, bis alle Zeichen als Knoten im Huffman-Baum vorhanden sind
4. Schreibe an jede linke Kante eine 0 und an jede rechte Kante eine 1. Lies für jedes Blatt den Pfad von der Wurzel bis zu dem jeweiligen Blatt ab und notiere so den Binärcode der einzelnen Zeichen
5. Ersetze alle Zeichen im Wort durch den jeweiligen Binärcode an den Blättern