Plan

- General Theory Recap
- Tips for the exam
- Github repo
- Questions
- Peer Grading 12.2

Sorting Algorithms (week 4 notes)

	Runtime	Space-Complexity
Bubble Sort	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n)$
QuickSort (very rare)	$O(n^2)/O(n \log n)$	$O(1)$
HeapSort	$O(n \log n)$	$O(1)$

Abstract Datatypes (ADT) (week 4 notes)

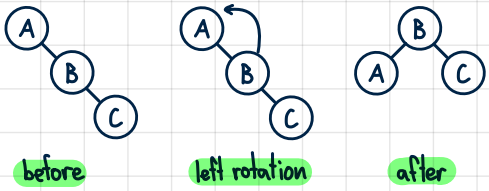
	insert	get	delete	insertAfter
Array	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Linked-List	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Doubly-Linked-List	$O(1)$	$O(n)$	$O(1)$	$O(1)$

if we know memory-location of object

AVL Trees (week 5 notes)

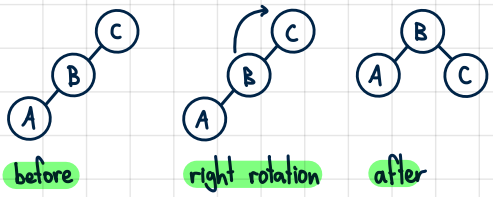
Left Rotation

- unbalanced due to right subtree of right subtree



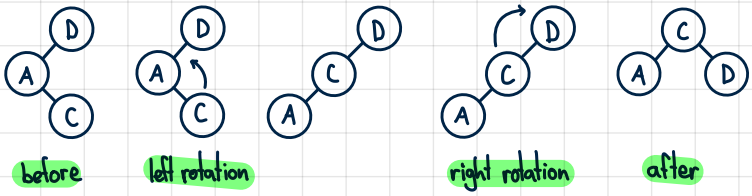
Right Rotation

- unbalanced due to left subtree of left subtree



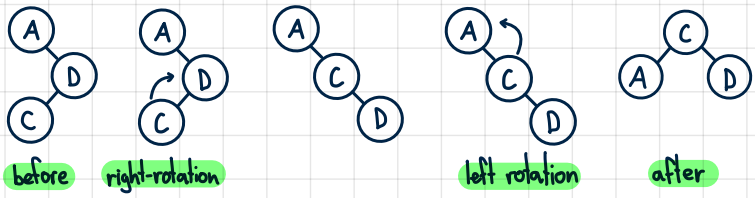
Left-Right Rotation

- combination of left and right rotation (in that order)
- unbalanced due to right subtree of left subtree



Right-Left Rotation

- combination of right and left rotation (in that order)
- unbalanced due to left subtree of right subtree



Graph Cheatsheet für Algorithmen und Datenstrukturen

Kenji Nakano, HS23, Stand 17.12.2023

Keine Garantie für Vollständigkeit oder Korrektheit

Definition Graph

Ein **Graph** ist ein Tupel $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ wobei

- $V :=$ Knotenmenge (vertices)
- $E :=$ Kantenmenge (edges)

jede Kante ist ein ungeordnetes Paar zweier Knoten $u \neq v$, $e = \{u, v\} \in E$ (Kurzform: uv)

Weg, Pfad, Zyklus

- **Weg:** Folge von benachbarten Knoten (engl. walk)
- **Pfad:** Weg ohne wiederholte Knoten
- **Zyklus:** Weg mit $v_0 = v_l$, $l \geq 2$ (engl. closed walk)

Die Länge eines Wegs bzw. Pfads ist die Anzahl an Kanten, nicht die Anzahl an Knoten

Begriffe

- u, v adjazent/benachbart $\Leftrightarrow e = \{u, v\} \in E$
- $e \in E$ inzident/anliegend zu $v \Leftrightarrow \exists u \in V$, so dass $e = \{u, v\}$
- $deg(u) =$ Knotengrad von u (Anzahl Nachbarn)
- u erreicht $v \Leftrightarrow \exists$ Weg zwischen u und v (engl. reachable)
  Äquivalenzrelation (symmetrisch, reflexiv, transitiv)
- Zusammenhangskomponente (ZHK): Äquivalenzklasse der "erreichbar"-relation (engl. connected component)
- Graph ist zusammenhängend $\Leftrightarrow$ es gibt gibt genau eine ZHK

Handschlag Lemma: $\sum_{v \in V} deg(v) = 2 \cdot |E|$

Eulerweg, Hamiltonpfad, Eulerzyklus

- **Eulerweg:** Weg welcher jede Kante **genau** einmal enthält (engl. Eulerian walk)
- **Hamiltonpfad:** Pfad der jeden Knoten **genau** einmal enthält
- **Eulerzyklus:** Zyklus welcher jede Kante **genau** einmal enthält

$\exists$ Eulerzyklus $\Leftrightarrow$ alle Knotengrade gerade und alle Kanten in einer ZHK

Algorithmus Eulertour / Eulerwalk

Euler(G):

- Input: Graph $G = (V, E)$
- Output: Liste Z mit Eulerzyklus, falls existiert.
- Laufzeit: $\mathcal{O}(m)$

EulerWalk(u):

- Input: Knoten $u \in V$
- Output: Keiner
- Laufzeit: $\mathcal{O}(m)$

Algorithm 1 Euler(G)

Require: Alle Kanten unmarkiert

1: $Z \leftarrow$ Leere Liste
2: EulerWalk(u_0) ▷ für $u_0 \in V$ beliebig
3: **return** Z

Algorithm 2 EulerWalk(u)

1: **for** $uv \in E$, nicht markiert **do**
2: markiere Kante uv
3: EulerWalk(v)
4: $Z \leftarrow Z \cup \{u\}$

Begriffe gerichtete Graphen

Der Graph $G = (V, E)$ ist definiert wie beim ungerichteten **ausser** Kanten sind geordnete Paare $e = (u, v) \in E$

- v ist Nachfolger von u
- u ist Vorgänger von v
- $deg_{in}(u)$ = Eingangsgrad
- $deg_{out}(u)$ = Ausgangsgrad
- Quelle u: $deg_{in}(u) = 0$
- Senke v: $deg_{out}(v) = 0$

DFS (Depth-First-Search)

DFS(G) (und Visit(u)):

- Input: Graph $G = (V, E)$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden)
- Laufzeit: $\mathcal{O}(n + m)$ (für Adjazenzlisten)

Algorithm 3 Visit(u)

```
1: pre[u] ← T; T ← T + 1
2: markiere u
3: for Nachvolger v von u, unmarkiert do
4:     Visit(v)
5: post[u] ← T; T ← T + 1
```

Algorithm 4 DFS(G)

```
1: T ← 1
2: alle Knoten unmarkiert
3: for u_0 ∈ V, unmarkiert do
4:     Visit(u_0)
```

BFS (Breadth-First-Search)

BFS(s):

- Input: Knoten $s \in V$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden, ähnlich wie DFS)
- Laufzeit: $\mathcal{O}(n+m)$

Algorithm 5 BFS(s)

```
1: Q ← {s}                                              ▷ Q ist eine Queue
2: enter[s] ← 0;   T ← 1
3: while Q ≠ ∅ do
4:     u ← dequeue(Q)
5:     leave[u] ← T;   T ← T + 1
6:     for (u, v) ∈ E, enter[v] nicht zugewiesen do
7:         enqueue(Q, v)
8:         enter[v] ← T;   T ← T + 1
```

Shortest-Path Algorithmen

Dijkstra(s):

- Input: Knoten $s \in V$
  - Graph darf nur nicht-negative Kantenkosten haben
- Output: $dist(s, v)$ für alle $v \in V$
- Laufzeit: $\mathcal{O}((m+n) \cdot log(n))$

Algorithm 6 Dijkstra(s)

```
1: d[s] ← 0;   d[v] ← ∞  ∀v ∈ V \ {s}
2: S ← ∅
3: H ← make-heap(V); decrease-key(H, s, 0)
4: while S ≠ V do
5:     v* ← extract-min(H)
6:     S ← S ∪ {v*}
7:     for (v*, v) ∈ E, v ∉ S do
8:         d[v] ← min{d[v], d[v*] + c(v*, v)}
9:         decrease-key(H, v, d[v])
```

Bellman-Ford(s):

- Input: Knoten $s \in V$
  - Graph darf auch negative Kanten (und negative Zyklen) enthalten.
- Output: $dist(s, v)$ für alle $v \in V$
- Laufzeit: $\mathcal{O}(m \cdot n)$
- Wenn man nach der $(n-1)$-ten Iteration nochmals eine weitere Iteration durchführt und die Werte sich nochmals verändern existiert ein gerichteter Zyklus mit negativem Totalgewicht

Algorithm 7 Bellman-Ford(s)

```
1: d[s] ← 0;   d[v] ← ∞ ∀v ∈ V \ {s}        ▷ 0-gute Schranken
2: for i ∈ {1, ... n − 1} do                ▷ Verbessere Schranken (n − 1)-mal
3:     for (u, v) ∈ E do
4:         d[v] ← min{d[v], d[u] + c(u, v)}
```

MST Algorithmen

Boruvka(G):

- Input: Graph $G = (V, E)$
- Output: Minimaler Spannbaum F
- Laufzeit: $\mathcal{O}((m+n) \cdot log(n))$ (wie Dijkstra)

Algorithm 8 Boruvka(G)

$F \leftarrow \varnothing$ ▷ sichere Kanten
while F nicht Spannbaum **do** ▷ $\leq log(n)$ Iterationen, $\mathcal{O}(m+n)$ pro Iteration
  $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
  $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
  $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

Idee: Konzentrieren uns auf eine ZHK

Prim(G, s):

- Input: Graph $G = (V, E)$ und $s \in V$
- Output: Minimaler Spannbaum F
- Laufzeit: $\mathcal{O}((m+n) \cdot log(n))$ (wie Dijkstra und Boruvka)

Algorithm 9 Prim(G, s) (allgemeine Form)

$F \leftarrow \varnothing$
$S \leftarrow \{s\}$ ▷ ZHK von s in F
while F nicht Spannbaum **do**
  $u^*v^* \leftarrow$ minimale Kante an S $\quad (u^* \in S, v^* \notin S)$
  $F \leftarrow F \cup \{u^*v^*\}$
  $S \leftarrow S \cup \{v^*\}$

Algorithm 10 Prim(G, s) (mit min-heap)

$H \leftarrow$ make-heap(V, ∞), $S \leftarrow \varnothing$
$\mathrm{d}[s] \leftarrow 0$; $\quad \mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
decrease-key$(H, s, 0)$
while $H \neq \varnothing$ **do**
  $v^* \leftarrow$ extract-min(H)
  $S \leftarrow S \cup \{v^*\}$
  for $v^*v \in E, \ \ v \notin S$ **do**
    $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], c(v^*, v)\}$ ▷ Unterschied zu Dijkstra
    decrease-key$(H, v, \mathrm{d}[v])$

Union-Find Datenstruktur

Idee: verwalte alle Knoten einer ZHK als Liste

Algorithm 11 Union-Find(G)

Implementierung:
MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$ ▷ $\mathcal{O}(n)$

SAME(u,v): teste ob rep[u] = rep[v] ▷ $\mathcal{O}(1)$

UNION(u,v): ▷ $\mathcal{O}(|\text{ZHK}(u)|)$
for $x \in$ members[rep[u]] **do**
    rep[x] $\leftarrow$ rep[v]
    members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

Idee: sichere Kanten sortiert nach Gewicht

Kruskal(G):

- Input: Graph $G = (V, E)$ und $s \in V$
- Output: Minimaler Spannbaum F
- Brauchen eine Union-Find Datenstruktur für effiziente Laufzeit
- Laufzeit: $\mathcal{O}(\underbrace{m \cdot log(m)}_{\text{Sortieren}} + \underbrace{n \cdot log(n)}_{\text{Union-Find}})$

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

$F \leftarrow \varnothing$
$UF \leftarrow$ MAKE(V) ▷ UF-Datenstruktur initialisieren
SORT(E) ▷ Sortiere Kanten nach Gewicht
for $uv \in E$, aufsteigend sortiert **do**
    if SAME(u,v) = false **then** ▷ u, v in verschiedenen ZHKs von F
        $F \leftarrow F \cup \{uv\}$
        UNION(u,v)

All pairs Shortest Path

Floyd-Warshall(G):

- Input: Graph $G = (V, E)$
- Output: Länge von beliebigem kürzestem Pfad.
- Laufzeit: $\mathcal{O}(n^3)$
- Teilproblem:

  d^i_{uv} = Länge von kürzestem $u - v$-Weg, der nur Zwischenknoten aus $\{1, \ldots, i\}$ benutzen darf

Algorithm 13 Floyd-Warshall(G)

```
for u ∈ V do
    d^0_uu ← 0                                   ▷ falls keine negativen Schleifen
    for v ∈ V \ {u} do
        if (u, v) ∈ E then
            d^0_uv ← c(u, v)
        else
            d^0_uv ← ∞
for i = 1 ... n do
    for u = 1 ... n do
        for v = 1 ... n do
            d^i_uv ← min{d^{i-1}_uv, d^{i-1}_ui + d^{i-1}_iv}
return d^n                                       ▷ n × n Resulatmatrix
```

Tips

Short Answer Questions

Multiple Choice: if you're absolutely unsure leave box empty

- keep your answers short

DP

- usually DP-dimensions can be derived from runtime
- draw a small example table
  ↳ check if your recursion works

Graph Exercise

- check for small hints (graph connected ⇒ $m \geq n-1$, etc.)
- read carefully what is asked
  ↳ pseudocode?
  ↳ runtime proof?
  ↳ correctness proof?
- if time is tight write bullet points
  ↳ correct idea often gives a lot of partial points

Preparation

- solve old exams
- read up on theory if you're stuck
- solve the uploaded CodeEx. exams
- if you need more DP-practice use LeetCode

If you have any questions during Lernphase send me an email or DM me on Discord