Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Stephan Heidinger

Algorithemn & Datenstrukturen
Wintersemester 2011/2012
1. Februar 2012

Find any errors? Please send them back, I want to keep them!

1 Einführung

1.1 Auswahlproblem

Ziel: „Bestimme das k. kleinste Element von n Elementen“

Spezialfälle: $k = \begin{cases} 1 & Minimumsuche \\ n & Maximumsuche \\ \lfloor \frac{n}{2} \rfloor & Median \end{cases}$

Verschiedene Ansätze:

1. Auswahl nach Sortieren
2. Wiederholte Minimumsuche
3. Aktualisierung einer vorläufigen Lösung
4. Nutzen von Standartbibliotheken

Bewertung ist schwer, bei 1 und 4 muss mehr über die Implementierung bekannt sein, bei allen Varianten muss zudem mehr über die Eingabe bekannt sein.

1.2 Maschinenmodell

Wie soll bewertet werden? Laufzeit in Sekunden? Hängt massgeblich ab von:

- Programmiersprache
- Rechner (Aufbau, Taktfrequenz, Speicher, ...)
- Eingabedaten

Müssen Bewertung unabhängig davon finden ⇒ **Zählen von *elementaren Schritten***

Random Access Machine

Maschinenmodell mit:

- endliche Zahl an Speicherzellen für Programm
- abzählbar endliche Zahl von Speicherzellen für Daten

- endliche Zahl von Registern
- arithmetisch-logische Einheit (ALU)

Anweisungen:

- Transportbefehle (Laden, Verschieben, Speichern)
- Sprungbefehle (bedingt, unbedingt; → Schleifen, Rekursionen)
- arithmetische und logische Vernküpfungen

1.3 Komplexität

Beschreiben Komplexität durch.

Laufzeit: Anahl Schritte (asymptotisch)

Speicherbedarf: Anzahl benutzter Speicherzellen

1.3.1 Definitionen

1. höchstens so schnell wachsen wie f, langsamer und gleich als f

$$\mathcal{O}(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ |g(n)| \leq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

2. mindestens so schnell wachsen wie f, schneller und gleich als f

$$\Omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ c \cdot |g(n)| \geq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

3. genauso so schnell wachsen wie f

$$\Theta(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c_1, c_2, n_0 > 0 \text{ mit} \\ c_1 \leq \frac{|g(n)|}{|f(n)|} \leq c_2 \text{ für alle } n > n_0 \end{array} \right\}$$

4. die gegenüber f verschwinden, langsamer als f

$$o(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{zu jedem } c > 0 \text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ c \cdot |g(n) \leq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

5. denen gegenüber f verschwindet, schneller als f

$$\omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \begin{array}{c} \text{zu jedem } c > 0 \text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ |g(n) \geq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

$$\log_a n \leq \sqrt{n} \leq n^2 \leq n^3 \leq 1,1^n \leq 2^n \leq n! \leq n^n$$

1.3.2 Satz

1. $g \in \mathcal{O}(f)$ genau dann, wenn $f \in \Omega(g)$
   $g \in \Theta(f)$ genau dann, wenn $f \in \Theta(g)$
2. $\log_b n \in \Theta(\log_2 n)$ für alle $b > 1$
   „Die Basis des Logarithmus spielt für das Wachstum keine Rolle“
3. $(\log_2 n)^d \in o(n^\varepsilon)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „Logarithmen wachsen langsamer als alle Polynomialfunktionen “
4. $n^d \in o((1+\varepsilon)^n)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „ Exponentielles Wachstum ist immer schneller als polynomielles “
5. $b^n \in o((b+\varepsilon)^n)$ für alle $b \geq 1, \varepsilon > 0$
   „ Jede Verringerung der Basis verlangt exponentielles Wachstum“
6. $\binom{a}{b} \in \Theta(n^k)$
7. $H_n := \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$ (harmonische Zahlen)
8. $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{c}^n\right) \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$ (Stirlingformel)

2 Sortieren

2.0.3 Gütekriterien

- Laufzeit
- Speicherbedarf
- Stabilität (kein Vertauschen der Reihenfolge schon sortierter Elemente)
- u.U. getrennte Betrachtung von Anzahl Vergleiche $C(n)$ und Anzahl Umspeicherungen $M(n)$

2.1 SelectionSort

2.1.1 Algorithmus

Idee: Man nehme pro Durchlauf das kleinste Element heraus und mache das solange, bis das Quellarray leer ist.

2.1.2 Laufzeit

$$\text{Anzahl Vergleiche:} \qquad C(n) = n-1+n-2+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2}$$

$$\text{Anzahl Umspeicherungen:} \qquad M(n) = 3\cdot(n-1)$$

Laufzeit liegt damit in $\Theta(n^2)$

2.1.3 Stabilität

Da weiter vorne stehende Elemente hinter gleiche andere vertauscht werden können: **nicht stabil**

2.2 Divide & Conquer: QuickSort

Divide & Conquer:
Zerteile Aufgabe in kleinere Aufgaben & Löse diese rekursiv.

Idee: Wähle ein Element p („Pivot“) und teile die anderen Elemente der Eingabe auf in :

M_1**:** die höchstens kleineren Elemente

M_2**:** die größeren Elemente

Sortierung von M erhält man nun durch Hintereinanderschalten von M_1, p, M_2

2.2.1 Algorithmus

Idee: Wähle Pivot und teile die Arrays dementsprechend auf.

2.2.2 Laufzeit

im besten Fall: $\Theta(n \log n)$

im schlechtesten Fall: $\Theta(n^2)$ (bereits sortierte Eingabe)

mittlere Laufzeit: $\Theta(n \log n)$

Eine zufällige Auswahl des Pivot führt zu einem Algorithmus, der im Mittel auch auf vorsortierten Eingaben schnell ist.

2.3 Divide & Conquer: MergeSort

MergeSort quasi umgekehrt zu QuickSort: triviale Aufteilung, linearer Aufwand bei Rekombination

2.3.1 Algorithmus

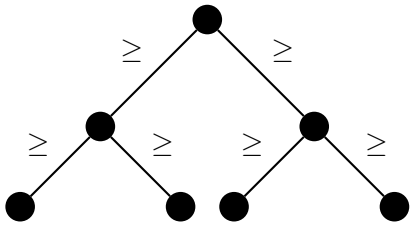
2.3.2 Laufzeit

Laufzeit ist in $\Theta(n \log n)$ (unabhängig von der Eingabe, dafür aber höherer Speicherbedarf)

2.4 HeapSort

2.4.1 Heap

Heap-Bedingung:
Für jeden Knoten gilt, dass der darin gespeicherte Wert nicht kleienr ist, als die beiden Werte in seinen Kindern.



Einfügen: Das neue Objekt wird hinten ins Array eingefügt, dann wird es solange mit seinem Vorgänger vertauscht, bis die *Heap-Bedingung* wiederhergestellt ist. **Laufzeit:** $\mathcal{O}(\log n)$

ExtractMax: Zum Entfernen wird das erste Element entfernt. Das letzte Element des Arrays wird an die Spitze gefüllt und „versickert“ nun nach unten. Dies wird mittels „heapify“ erreicht.

heapify: Lässt aktuelles Element nach unten „sickern“, indem es den Platz mit dem Größeren der Kinder vertauscht.

2.4.2 Algorithmus

Idee: Der Algorithmus teilt sich in 2 Phasen:

1. Herstellen der Heap-Eigenschaft:
   Starten bei $\lfloor \frac{n}{2} \rfloor$, rufen hier `heapify` auf und gehen dann zu $\lfloor \frac{n}{2} \rfloor, \ldots, 1$
2. Abbau des Heap (Maximumsuche und Vertauschen nach Hinten:
   Um Speicherplatz zu sparen wird im selben Array gearbeitet. Dazu wird immer das erste Element mit dem letzten Element des verleibenden Heapes vertauscht und dieses neue dann „versickert“ mittels `heapify`. Danach wir der Heap um 1 verkleinert.

2.4.3 Laufzeit

Die Laufzeit ist $\mathcal{O}(n \log n)$.

2.5 Untere Laufzeitschranke

Frage: Wie schnell kann man überhaupt Sortieren (bzw etwas machen). Sortieren kann auch als Entscheidungsbaum dargestellt werden. Anhand des ersten Vergleiches folgen weitere Vergleiche, abhängig vom ersten Vergleich. Das sind $n!$ Blätter und somit:

$$\begin{aligned} \log n! &= \log[n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1] \\ &= \log\left[\left(\frac{n}{2}^{\frac{n}{2}}\right)\right] \qquad = \frac{n}{2} \cdot \log \frac{n}{2} \in \Omega(n \log n) \end{aligned}$$

Jedes allgemeine Sortierverfahren benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

2.6 Sortierverfahren für spezielle Universen

2.6.1 BucketSort

Annahme: Wollen nur reelle Zahlen sortieren. O.B.d.A. gilt $\mathcal{U} = (0, 1]$. Für die Zahlen $M = \{a_1, \ldots, a_m\}$ werden nun n Buckets erstellt und jede Zahl in den entsprechenden Bucket geworfen. Die einzelnen Buckets werden dann mit einem anderen Verfahren sortiert. Daher ist BucketSort ein *Hüllensortierverfahren.*

Laufzeit Die mittlere Laufzeit von Bucketsort ist in $\mathcal{O}(n)$

2.6.2 CountingSort

Annahme: Wollen ganzzahlige Zahlen sortieren: $\mathcal{U} \subseteq \mathbb{N}_0 = \{0, \ldots, n\}$. O.B.d.A. gilt $M = \{0, \ldots, k-1\}$. Es wird nun für jede Zahl $1, \ldots, n$ ein Bucket erstellt und die Zahlen dort einsortiert. Danach werden die nun sortierten Zahlen wieder zurück ins Ursprungsarray geführt. Damit die evtl Zuordnung zu Daten erhalten bleibt, braucht es noch ein Zusatzarray, aus welchem diese Zuordnung nachher wieder abgelesen werden kann.

Laufzeit Die Laufzeit von CountingSort ist in $\mathcal{O}(n+k)$.

Stabilität Nicht stabil.

2.6.3 RadixSort

Angenommen wir wollen Zahlen sortieren aus $\mathcal{U} = \{0, \ldots, d-1\}$ und wir haben ein stabiles Sortierverfahren für die d-äre Darstellung. Dann sortieren wir nach den Stellen, von hinten nach vorne. s ist die Anzahl der Stellen. Als Sortierverfahren bietet sich CountingSort an.

Laufzeit Die Laufzeit ist in $\Theta(s \cdot (n+d))$. Ein großes d (z.B. $d = 256 \Rightarrow$ byteweise Aufteilung) lohnt sich, da dann s klein wird.

2.6.4 RadixExchangeSort

Annahme: Zahlen in Binärdarstellung, Partitionierung aufgrund der der jeweils aktuellen Binärziffer

Laufzeit Die Laufzeit ist $\Theta(s \cdot n)$. Besonders geeignet, wenn s klein im Verhältnis zu n.

Stabilität Nicht stabil.

2.7 Gegenüberstellung

Algorithmus	Lauf-zeit-klasse worst	-zeit-klasse average	-klasse best	Speicher	stabil	Einschränkung
SelectionSort	n^2	n^2	n^2	$\Theta(1)$	✘	keine
QuickSort	n^2	$n \log n$	$n \log n$	$\mathcal{O}(n)$	✘	keine
MergeSort	$n \log n$	$n \log n$	$n \log n$	$\mathcal{O}(n)$	✔	keine
HeapSort	$n \log n$	$n \log n$	n	$\Theta(1)$	✘	keine
untere Schranke	$n \log n$	$n \log n$	n	n.a.	n.a.	keine
BucketSort	$n \log n$	n	n	$\Theta(1)$	✔	reelle Zahlen aus $(0, 1]$
CountingSort	$n+k$	$n+k$	$n+k$	$\Theta(n+k)$	✔	ganze Zahlen aus $(0, k-1)$
RadixSort	$s \cdot (n+d)$	$s \cdot (n+d)$	$s \cdot (n+d)$	$\Theta(n+d)$	✔	d-äre Zahlen, Wortlänge s
RadixExchangeSort	$s \cdot n$	$s \cdot n$	$s \cdot n$	$\mathcal{O}(n)$	✘	Binärzahlen, Bitlänge s

3 Suchen

Dictionary:

Schlüssel: eindeutige Kennung

Element: zu einem Schlüssel gehörender Datensatz

3.1 Lineare Suche

Suchen in einem unsortierten Array/einer unsortierten Liste.

3.1.1 Laufzeit

Lineare Suche benötigt im mittleren Fall $\mathcal{O}(n)$. Kommt der Schlüssel nicht vor, muss das ganze Array/die ganze Liste durchlaufen werden.

3.2 Selbstanordnende Folgen

Annahme: Die Reihenfolge der Liste darf geändert werden. Anfragehäufigkeiten sind unbekannt. Wir betrachten 3 Strategien:

- MF (move to front): Der Schlüssel auf den zugegriffen wurde, wird nach vorne verschoben.
- T (tranpose): Der Schlüssel, auf den gerade zugegriffen wurde, wird mit seinem Vorgänger vertauscht.
- FC (frequency count): Die Liste wird entsprechend einem Zähler sortiert, der die Zugriffshäufigkeiten angibt.

Empirisch zeigt sich, dass T schlechter ist als MF, FC und MF sind ähnlich, MF aber manchmal besser.

$C_A(S)$	Kosten für Zugriffe S
$F_A(S)$	kostenfreie Vertauschungen von benachbarten Schlüsseln an den untersuchen Positionen
$X_A(S)$	konstenpflichtige Vertauschungen

Für jeden beliebigen Algorithmus A zur Selbstanordnung gilt für jede Folge S von Zugriffen:

$$C_{MF} \leq 2 \cdot C_A(S) + X_A(S) - F_A(S) - |S|$$

3.3 sortierte Arrays, binäre Suche

Annahme: Die Elemente sind sortiert. Wir betrachten immer das mittlere des verbleibenden Arrays und können damit die Hälfte der verbleibenden ausschliessen, da der Schlüssel größer oder kleiner sein muss (oder der gesuchte)

Laufzeit Die Laufzeit für binäre Suche beträgt $\Theta(\log n)$ um einen Schlüssel zu finden, bzw herauszufinden, ob er nicht enthalten ist.
Bei Prozessoren mit Pipelining/Multithreading kann eine ungleiche Aufteilung sinnvoller sein.

3.4 geordnete Wörterbücher

geordnetes Wörterbuch: so angeordnet, dass zu jeder Zeit alle Paare aus Schlüssel und Element mit linearem Aufwand in Sortierreihenfolge ausgegeben werden können. (Vorraussetung ist, dass über den Schlüsseln eine Ordnung $\leq$ besteht)

3.4.1 binärer Suchbaum

Speicherung:

•	Node
`node`	parent
`node`	left, right
`item`	item

Suchbaumeigenschaft:

$$w.key < v.key \qquad \forall w \in L(v)$$
$$w.key > v.key \qquad \forall w \in R(v)$$

wobei $L(v)$ den linken und $R(v)$ den rechten Teilbaum von v bezeichnet.
Ausgabe erfolt mittels eines `inorder`-Suchlaufes (Links,Mitte,Rechts)

ADT dictionary:

`find`(k)Element mit Schlüssel k finden
`search`(T, k)suche k im (Unter)Baum T
`insert`(a, k)einfügen von Element a mit Schlüssel k
`remove`$(k)k$Element mit Schlüssel k Entfernen

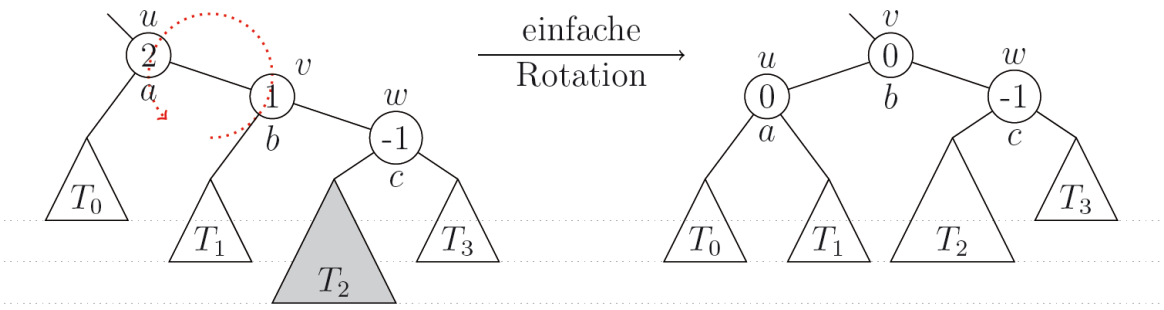
3.5 AVL-Bäume

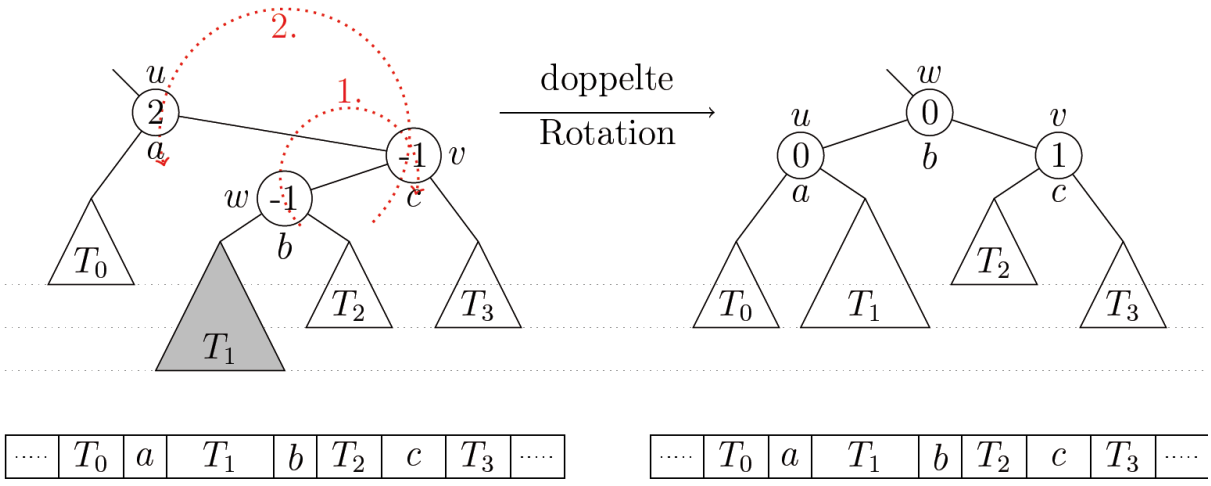
Um die Entartung von Bäumen zu verhindern, wird eine zusätzliche Balanciertheitseigenschaft gefordert.

Balanciertheitseigenschaft: Für jeden (inneren) Knoten eines Binärbaums gilt, dass die Höhe der Teilbäume der beiden inneren Kindern sich um höchstens 1 unterscheidet.

Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Die Operationen `find` und `search` bleiben unverändert. Bei `insert` (nur in Blätter) und `remove` muss jedoch die Balanciertheitseigenschaft beachtet werden. Dazu werden Rotationen eingefügt, die Knoten umsortieren, um diese Eigenschaften wiederherzustellen (S.49 im Script).





3.6 Rot-Schwarz-Bäume

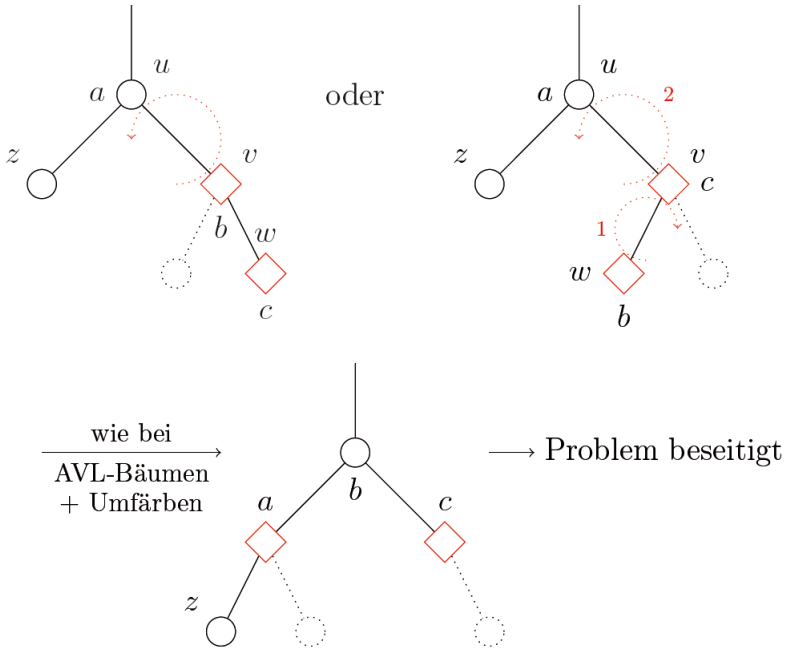
Alternativ kann man bestimmte Knoten von der Eigenschaft für einen perfekten Binärbaum ausnehmen, also „Färben“:

schwarz: normaler Knoten
rot: Ausgleichsknoten (für Höhenunterschiede)
Wurzelbedingung: Die Wurzel ist schwarz.
Farbbedingung: Die Kinder von roten Knoten sind schwarze Knoten (oder nil).
Tiefenbedingung: Alle Blätter haben die gleiche schwarze Tiefe (definiert als die Anzahl schwarze Vorfahren -1)

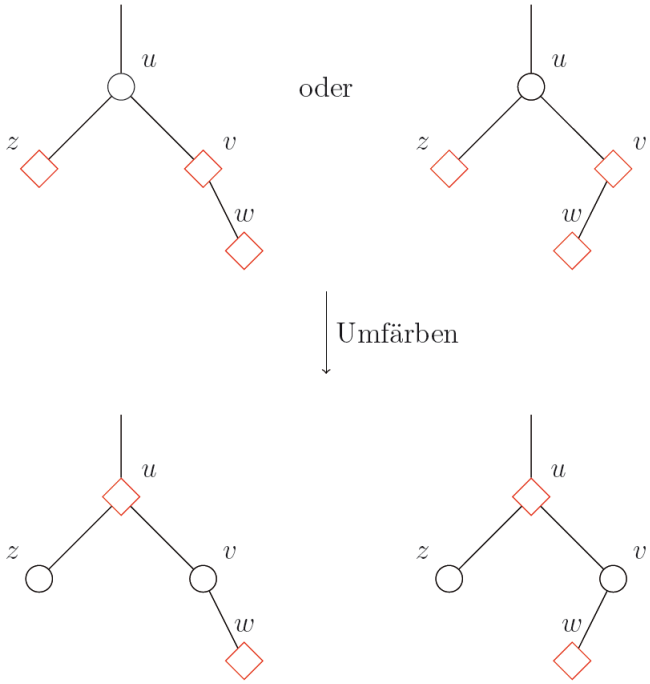
Die Höhe eines rot-schwarz Baumes mit n Knoten ist $\Theta(\log n)$

Beim Einfügen wird das neue Blatt rot (ausser, wenn das neue Blatt die Wurzel ist). Ist der Elternknoten des neuen Blattes jedoch bereits rot, ist die Farbbedingung verletzt. Sollte nach dem Rotieren bei u Doppelrot auftreten, wird solande nach oben iteriert, bis man die Wurzel erreicht.

Fall 1: Der Geschwisterknoten z von v ist schwarz:



Fall 1: Der Geschwisterknoten z von v ist rot:



Löschen von AVL-Bäumen (S.53-55 im Script)

3.7 B-Bäume

Ist der Index zu groß, müssen Werte evtl weit verstreut abgelegt werden.

Vielwegbaum: speichert pro Knoten mehrere Schlüssel in aufsteigender Reihenfolge. Zwischen je zwei Schlüsseln, sowie vor dem Ersten und nach dem Letzten wird in Kind/Teilbaum angehängt.

Die Zahl der Kinder ist immer Anzahl Schlüssel -1
B-Baum der Ordnung d, $d \geq 2$
Wurzelbedingung: Die Wurzel hat mindestens 2 und höchstens d Kinder.
Knotenbedingung: Jeder andere Knoten hat mindestens $\lceil \frac{d}{2} \rceil$ und höchstens d Kinder.
Schlüsselbedingung: Jeder Knoten mit i Kindern hat $i-1$ Schlüssel.
Tiefenbedingung: Alle Blätter haben dieselbe Tiefe.

Ein B-Baum der Ordnung d miz n Knoten hat Höhe $\Theta(\log_d n)$

`insert`: Suche das Blatt, in das eingefügt werden soll. Wieviele Schlüssel hat das Blatt?

$< d-1$ einfügen, fertig

$= d-1$ Überlauf: teile die d Schlüssel auf in die kleineren $\lfloor \frac{d}{2} \rfloor$ und die größeren $\lceil \frac{d}{2} \rceil + 1$, für die zwei neue Knoten erzeugt werden. Das mittlere $(\lfloor \frac{d}{2} \rfloor)$-te wird im Elternknoten eingefügt, evtl Überlauf im Elternknoten durch nach oben iterieren beheben.

`remove`: Suche den Knoten, aus dem ein Schlüssel entfernt werden soll. Wieviele Schlüssel enthält er?

$> \lceil \frac{d}{2} \rceil - 1$ löschen, fertig

$= \lceil \frac{d}{2} \rceil - 1$ Unterlauf

1. Knoten ist innerer Knoten (d.h. kein Blatt)
   Ersetze Schlüssel durch Inorder-Vorgänger (oder Nachfolger), d.h. letzten Schlüssel im rechtesten Blatt des vorangeheneden Teilbaumes.
2. Knoten ist Blatt
   Falls ein direkter rechter oder linker Geschwisterknoten mehr als $\lceil \frac{d}{2} \rceil - 1$ Schlüssel enthält, verschiebe den Letzten bzw Ersten von dort in den gemeinsamen Vorgänger und den trennenten Schlüssel von vorher in den Knoten mit Unterlauf.
   Anderfalls Knoten mit einem der direkten Geschwisterknoten und dem trennenden Schlüssel aus dem Vorgänger verschmelzen.

4 Streuen

Betrachten ungeordnete Wörterbücher. Schlüssel ohne Sortierreihenfolge gespeichert.

Bezeichnungen:	
(Schlüssel-)Universum	$\mathcal{U} \subseteq \mathbb{N}_0$
Schlüsselmenge	$K \subseteq \mathcal{U},\ \lvert K \rvert = n$
Hashtabelle	$H[0, \ldots, m-1]$ (Array von Zeigern)

Eine Hashfunktion $h : \mathcal{U} \to \{0, \ldots, m-1\}$ weisst jedem Schlüssel $k \in \mathcal{U}$ seine Speicherstelle $H[h(k)]$ zu. Im Optimalfall ist jede Stelle in der Hashtabelle nur einfach besetzt ($\Rightarrow$ Einfügen, Suchen, Löschen in $\Theta(1)$).
Beispiele für Hashfunktionen:

- $h(k) = k \bmod m$ für Primzahl m (Divisions- /Kongruenzmethode)
- $h(k) = \lfloor m \cdot (k\alpha - \lfloor k\alpha \rfloor) \rfloor$, zBsp für $\alpha = \phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0,61803$ (Multiplikationsmethode)

4.1 Kollissionen

Im Allg. kommt vor: $h(k_1) = h(k_2)$ für einige $h_1, h_2 \in K \Rightarrow$ heisst *Kollision*, k_1, k_2 heissten *Synonym*.

$p_{m,n} := P(\text{keine Kollision}) = 1 \cdot \frac{m-1}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m}$
Schlüssel werden durch h gleichmässig gestreut, d.h. alle Positionen gleich wahrscheinlich. (Bsp. Geburtstagsparadoxon)
Erwartete Anzahl Kollisionen:

Sei $X_{ij} = \begin{cases} 1 & \text{falls } h(k_i) = h(k_j)_{\text{(Also bei Kollision)}} \\ 0 & sonst \end{cases}$
Bei Gleichmässiger Streuung: $E(X) = E\left(\sum_{i<j} X_{ij}\right) = \binom{n}{2} \cdot \frac{1}{m}$
Im Allgemeinen: $E(X) \geq 1 \Leftrightarrow n(n-1) \geq 2m$
Belegungsfaktor (load factor): $\beta = \frac{n}{m}$

Erwartete Anzahl Leerfelder: $m - m \cdot e^{-\beta} = m(1 - e^{-\beta})$

4.2 Kollisionsbehandlung

4.2.1 Verkettung

In der Hashtabelle wird eine Liste von Elementen angelegt. Wenn mehrere Zugriffe auf Zeiger erfolgen, heisst dies Sondieren. Im Mittel sind $n = m \ln m$ Einfügungen nötig, um alle Felder zu füllen, dann ist $\beta = \frac{m \ln m}{m} = \ln m > 1$

4.2.2 Open Hashing

Idee: In jedem Feld wird nur ein Element gespeichert. $\Rightarrow$ Bei Kollision muss neuer Platz gefunden werden. Wähle dazu eine Folge $d_1, d_2, \ldots, d_i \in \mathbb{Z}$ und versuche

$$H[(h(k) + d_i) \bmod m],\ i = 1, 2, \ldots$$

Für (d_i) gibt es verschiedene Wahlen:

lineares Sondieren	$d_i = i$
quadratisches Sondieren	$d_i = i^2$
double Hashing	$d_i = i \cdot h'(k)$, wobei $h' : \mathcal{U} \to \{1, \ldots, m-1\}$ eine zweite Hashfunktion mit $h'(k) \neq 0\ \forall k \in \mathcal{U}$ und m prim ist.

4.2.3 Kollisionsvermeidung

Ziel: Gleichmässiges Streuen durch Hash-Funktion
Problem: Wissen nicht, welche Schlüssel gespeichert werden sollen.
Idee: Wähle aus einer Menge von Hashfunktionen $\mathcal{H} \subseteq \{h : \mathcal{U} \to \{0, \ldots, m-1\}\}$ zufällig eine aus.

Eine Familie von Streufunktionen heisst universell, falls

$$|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}\ \forall k_1, k_2 \in \mathcal{U}$$

Bei zufälliger Wahl sind wir berechtigt $P(h(k_1) = h(k_2)) = \frac{1}{m}$ anzunehmen.

Das ist mir jetzt grad zu blöd die Formeln weiter abzutippen, Seite 68ff

4.2.4 Bloom-Filter (aus der Übung)

Bloom-Filter besteht aus m Bits, die zu Beginn 0 sind, k verschiedenen Hash-Funktionen mit Wertebereich $\{0, \ldots, m-1\}$

Einfügen: Berechne alle k Hashwerte und setze die entsprechenden Bits auf 1.

Suchen ob drin: Berechne k Hashwerte

- wenn alle Bits 0, auf keinen Fall enthalten
- wenn alle Bits 1, mit sehr großer Wahrscheinlichkeit enthalten
- wenn einige Bits 1, evtl vorhanden

5 Dynamische Programmierung: Ausrichten

Idee: Suchen nach „ähnlichen“ Wortketten.

- Σ ist Alphabet
- $s = a_1 a_2 \cdots a_n \in \Sigma^* = \bigcup \Sigma^k$ heisst „Wort“ oder „Sequenz“ der Länge $|s| = n$
- ε ist das leere Wort, $|\varepsilon| = 0$
- Teilsequenz s' von $s = s = a_1 a_2 \cdots a_n$ erfüllt $s' = a_i \cdots a_j$ für $1 \leq i, j \leq n$ (im Unterschied zum Teilwort, bei dem auch Zeichen ausgelassen werden können.

Um zwei Wörter zu vergleichen, gibt es drei Operationen. Ähnlichkeit ist die minimale Anzahl Operationen

Substitution: Ein Zeichen wird durch ein anderes ersetzt.

Einfügung: Ein Zeichen wird in einem Wort hinzugefügt.

Löschung: Ein Zeichen wird in einem Wort entfernt

Ein Paar heisst $\overline{s}, \overline{t} \in \Sigma^* \times \Sigma^*$ Ausrichtung von $s, t \in \Sigma^*$, falls

- $\overline{s}|_\Sigma = s, \overline{t}|_\Sigma$ (Weglassen aller Leerzeichen ergibt Ausganssequenz)
- $|\overline{s}| = |\overline{t}|$ (Beide Sperrungen gleich lang)
- Bewertung und Editierbarkeit Script S74

Der Ausrichtalgorithmus aus der Übung

Eine optimale Ausrichtung kann in $\mathcal{O}(nm)$ mit $\mathcal{O}(\min\{n, m\})$ Platz bestimmt werden.

6 Graphen

6.1 Definitionen

Das Paar $G = (V, E)$ aus den Mengen V und $E \subseteq \binom{V}{2}$ heisst (endlicher, ungerichteter) Graph, Elemente aus V heissen Knoten, Elemente aus E heissen Kannten. Allgemein: $n = n(G) = |V|$ und $m = m(g) = |E|$.
Zwei Knoten $u, v \in V$ heissen adjazent (benachbart), falls es eine Kante $\{u, v\} \in E$ gibt.
Die Adjazenzmatrix $A(G) = (a_{v,w \in V}$ mit

$$a_{v,w} = \begin{cases} 1 & \text{falls } \{v, w\} \in E \\ 0 & sonst \end{cases}$$

Die Inzidenzmatrix $I(G) = (i_{v,e})_{v \in V \wedge e \in E}$ mit

$$i_{v,e} = \begin{cases} 1 & \text{falls } v \in e \\ 0 & sonst \end{cases}$$

Ein Knoten $v \in V$ und eine Kante $e \in E$ heissen inzident, falls $v \in e$.
Die Menge der zu v adjazenten Knoten heisst Nachbarschaft und deren Kardinalität Grad d_G
Ein Teilgraph $G' = (V', E')$ mit $V' \subseteq V, E' \subseteq E$ heisst Teilgraph von G, G enthält G'.
Ein Teilgraph heisst aufspannend, wenn er alle Knoten enthält.
Gibt es zu zwei Graphen G_1, G_2 eine isomorphe Abbildung, so heissen G_1 und G_2 isomorph
Ein (s, t)-Weg der Länge k ist vorhanden, wenn eine Verbindung mit k-Kanten zwischen s und t besteht. Der kürzeste Weg heisst Abstand oder Distanz.
Ein Kreis ist ein Weg, der denselben Start- wie Endknoten hat.
Ein Graph heisst zusammenhängend, wenn es zu je zwei Knoten $s, t \in V$ einen Weg gibt. ($\Rightarrow$ Zusammenhangskomponenten)
Für alle Graphen $G = (V, E)$ gilt: $\sum_{v \in V} d_G(v) = 2m$. Die Anzahl der Knoten ungeraden Grades ist gerade. Es gibt immer zwei Knoten mit gleichem Grad.

6.2 Bäume und Wälder

Ein zusammenhängender Graph ohne Kreis heisst Baum. Ein Graph, dessen Zusammenhangskomponenten Bäume sind, ist ein Wald.
Für jeden Graphen $G = (V, E)$ gilt: $m \geq n - \kappa(G)$, Bei Gleichheit ist G ein Wald.
Für einen Graphen sind folgende Aussagen äquivalent:

1. G ist ein Baum.
2. Zwischen je zwei Knoten in G existiert genau ein Weg.
3. G ist zusammenhängend und hat $n - 1$ Kanten.
4. G ist minimal zusammenhängend.
5. G ist maximal kreisfrei.

Jeder zusammenhängende Graph enthält einen Baum. Die Anzahl $\mu(G) = m - n + \kappa(G)$ heisst zyklomatische Zahl.
Kanzenzugdefinitionen usw S86
Eine Tour heisst Eulertour falls sie jede Kante genau einmal enthält. Ein Graph enthält genau dann eine Eulertour, wenn alle Kanten geraden grad haben.

6.3 Tiefensuche

Idee: Knotenstack speichert noch zu bearbeitende Knoten. Wir gehen zuerst über Nachbarn in die Tiefe, wenn keine Knoten mehr vorhanden sind, gehen wir zum letzten bereits besuchten Knoten zurück, der noch unbesuchte Nachbarn hat.

6.3.1 Laufzeit

Die Laufzeit ist in $\mathcal{O}(n + m)$

6.4 Breitensuche

Idee: Knotenwarteschlange (Queue), wir besuchen von einem Knoten erst alle Nachbarn. Jeder markierte Knoten wird in die Queue gelegt. Dann wird jeweils der vorderste entnommen und von diesem alle Nachbarn besucht.

6.4.1 Laufzeit

Die Laufzeit ist in $\mathcal{O}(n + m)$

6.5 Kürzeste Wege

Betrachten nun gerichtete Graphen. Kanten werden durch Gewichte unterschiedlich bewertet (labeled).

Single-Source Shortes-Path Problem (SSSP):
geg: gerichteter Graph $G = (V, E, \lambda)$ mit Kantenlängen/-gewichten $\lambda : E \to \mathbb{R}$
ges: Kürzeste Wege zu allen Knoten

6.5.1 Algorithmus von Dijkstra

Idee: Wir optimieren lokal und hangeln uns dann im Graph weiter vor.
Dazu wird jeweils der aktuell kürzeste Pfad zu jedem Knoten gespeichert. Wenn ein Knoten noch nicht besuchte inzidente Kanten hat, wird er in eine Warteschlange gelegt. Für den vordersten Knoten in der Warteschlange wird nun überprüft, welche inzidente Kante den kürzesten Weg zum nächsten Knoten bildet, dieser wird mit dem aktuell kürzesten Weg zum gerade Betrachteten Knoten addiert und dem neuen Knoten gutgeschrieben. Wenn der neuen Knoten bereits einen Weg hat, wird der kürzere der beiden genommen.
Einschränkung: funktioniert nur mit nicht-negativen Kantengewichten.

Laufzeit Je nach verwendeter Struktur zur Speicherung des Graphen bis zu $\mathcal{O}(m + n \log n)$ Fibonacci-Heap bzw $\mathcal{O}(n + m \log n)$ Heap aus Kapitel 2

6.5.2 Algorithmus von Bellman/Ford

Vorteil: Kann auch mit negativen Kantengewichten umgehen.
Nachteil: Benötigt $\mathcal{O}(nm)$
Nicht möglich bei: enthaltene Zykel negativer Länge sonst würde er unendlich diesen Zykel nehmen...
Führt Relaxationen in schematisch fester Reihenfolge durch.

7 amortisierte Analyse

Idee: wir wollen eine schärfere Laufzeitschranke finden, als wir dies normal könnten. Verwenden dazu die „Bank-Account-Method“, betrachten die Gesamtlaufzeit, nicht die einzelnen Operationen. Werden bei zeitlich früherer Operation mehr „Zeit verbuchen“ als nötig, um dadurch folgende Operationen günstiger zu bekommen.

7.1 Beispiel: Binärzähler

	reale Kosten
0	
1	1
10	2
11	1
100	3
101	1

Kosten: ⇒ sehr unterschiedliche Kosten

			5	$\curvearrowright 1 \curvearrowright 10$
		5	4	$\curvearrowright 11$
		5	8	$\curvearrowright 100$
	5	4	7	$\curvearrowright 101$
	5	4	11	$\curvearrowright 111$
	5	8	15	

	normale Analyse	amortisierte Analyse
eine Einfügeoperation	$\mathcal{O}(\log n)$	
n Operationen	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$