1 Komplexität und Laufzeiten

1.1 Definitionen

D_n: Menge aller Eingaben der Länge n
$t(I)$: Benötigte Anzahl elementarer Operationen für Eingabe I
$Pr(I)$: Wahrscheinlichkeit, dass Eingabe I auftritt

1.2 Worst-, Best- und Average-Case-Laufzeit

W(n) = max $\{t(I)|I \in D_n\}$
B(n) = min $\{t(I)|I \in D_n\}$
A(n) = $\sum_{I \in D_n} Pr(I) * t(I)$

1.3 Asymptotische Betrachtung

Koeffizienten und Terme niedriger Ordnung werden ignoriert

1.4 L'Hospital

Seien f, g: $\mathbb{N} \to \mathbb{R}$ differenzierbare Funktionen mit $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$
Dann gilt: $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{g'(n)}{f'(n)}$

1.5 Die Klasse Gross-O

$O(f)$ ist die Menge von Funktionen, die **nicht schneller** als die Funktion f wachsen.
Diese Eigenschaft gilt ab einer Konstanten n_0, Werte unter n_0 werden vernachlässigt.
$g \in O(n^2)$ sagt mehr aus als $g \in O(n^3)$.

Beispiel: $g \in O(f)$ heißt, dass $c * f(n)$ eine obere Schranke für $g(n)$ ist.

$O(cf(n)) = O(f(n))$
$O(f(n)) + O(g(n)) = O(f(n) + g(n))$
$O(f(n)) + O(g(n)) = O(\text{max } \{f(n), g(n)\})$
$O(f(n))O(g(n)) = O(f(n)g(n))$
$f(O(1)) = O(1)$

$\sum_{k=1}^{n} O(f(k)) \subseteq O(nf(n))$, wenn $f(n)$ monoton steigt

1.6 Die Klasse Gross-Omega

$\Omega(f)$ ist die Menge von Funktionen, die **nicht langsamer** als die Funktion f wachsen.
Diese Eigenschaft gilt ab einer Konstanten n_0, Werte unter n_0 werden vernachlässigt.

$g \in \Omega(n^2)$ sagt mehr aus als $g \in \Omega(n)$.

Beispiel: $g \in \Omega(f)$ heißt, dass $c * f(n)$ eine untere Schranke für $g(n)$ ist.

1.7 Die Klasse Gross-Theta

$\Theta(f)$ ist die Menge von Funktionen, die **genauso schnell** wie die Funktion f wachsen.
Diese Eigenschaft gilt ab einer Konstanten n_0, Werte unter n_0 werden vernachlässigt.

Beispiel: $g \in \Theta(f)$ heißt, dass $c_1 * f(n)$ eine untere Schranke und $c_2 * f(n)$ eine obere Schranke für $g(n)$ ist.

$g \in \Theta(f)$ wenn $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ für ein $0 < c < \infty$
$g \in \Theta(f)$ gdw. $g \in O(f)$ und $g \in \Omega(f)$

1.8 Die Klasse Klein-O und Klein-Omega

$o(f)$ ist die Menge der Funktionen, die **echt langsamer** als f wachsen.
$\omega(f)$ ist die Menge der Funktionen, die **echt schneller** als f wachsen.

$f \in o(g)$ gdw. $g \in \omega(f)$

1.9 Elementare Eigenschaften

Reflexivität: $f \in O(f)$, $f \in \Omega(f)$, $f \in \Theta(f)$
Transitivität: Aus $f \in O/\Omega/\Theta(g)$ und $g \in O/\Omega/\Theta(h)$ folgt $f \in O/\Omega/\Theta(h)$.
Symmetrie von Θ: $f \in \Theta(g)$ gdw. $g \in \Theta(f)$
Beziehung zwischen O und Ω: $f \in O(g)$ gdw. $g \in \Omega(f)$

1.10 Platzkomplexität

Wir definieren $S(n)$ als Platzkomplexität.
Bei der Speicherung eines Liedes ist $S(n) \in O(n)$ und $S(n) \in \Omega(1)$.

2 Suchalgorithmen

2.1 Bilineare Suche

Array wird aus beiden Richtungen „gleichzeitig“ durchsucht.
Zeitkomplexität wie bei linearer Suche $O(n)$.

2.2 Binäre Suche

Sortiertes Array E mit n Elementen, gesucht ist das Element K.
Wir beginnen in der Mitte und **halbieren** den Suchraum in jedem Durchlauf.
Zeitkomplexität ist $O(\log\ n)$, allerdings ist die Initialisierung mit $O(n\ \log\ n)$ aufwändig.

3 Rekursionsgleichungen

3.1 Mastertheorem

$T(n) = b * T(\frac{n}{c}) + f(n)$ mit $b \geq 1$ und $c > 1$
Anzahl der Blätter im Rekursionsbaum: n^E mit $E = \frac{log(b)}{log(c)} = \log_c(b)$

Fall 1
Bedingung: $f(n) \in O(n^{E-\epsilon})$ für ein $\epsilon > 0$
Es folgt: $T(n) \in \Theta(n^E)$

Fall 2
Bedingung: $f(n) \in \Theta(n^E)$
Es folgt: $T(n) \in \Theta(n^E\ *\ \log\ n)$

Fall 3
1. Bedingung: $f(n) \in \Omega(n^{E+\epsilon})$ für ein $\epsilon > 0$
2. Bedingung: $f(\frac{n}{2}) \leq \frac{d}{4} * f(n)$ für ein $d < 1$ und ein hinreichend großes n
Es folgt: $T(n) \in \Theta(f(n))$

4 Sortieren

4.1 Grundlegendes

Ein Sortieralgorithmus ist **stabil**, wenn er die Reihenfolge der Elemente, deren Sortierschlüssel gleich sind, aufrecht erhält.

4.2 Insertionsort

Unsortierte Elemente werden in sortiertes Array vor dem nächstgrößeren Element eingefügt.
Insertionsort ist im Worst- und Average-Case quadratisch, im Best-Case linear.

4.3 Divide-and-Conquer-Algorithmen

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.
Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

4.4 Mergesort

Array wird in zwei möglichst gleich große Hälften geteilt, die durch rekursive Mergesort-Aufrufe sortiert werden.
Es ist $W(n) = B(n) = A(n) \in \Theta(n * log(n))$.
Speicherbedarf $\Theta(n)$ für die Kopie des Arrays und $\Theta(log(n))$ für die Verwaltung der Rekursion.

4.5 Heaps

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält. Für **Max-Heaps** gelten folgende Bedingungen:

- Der Schlüssel eines Knotens ist stets mindestens so groß wie die Schlüssel seiner Kinder.
- Alle Ebenen (mit Ausnahme der untersten) sind komplett gefüllt.
- Die Blätter befinden sich damit auf einer (oder zwei) Ebene(n).
- Die Blätter der untersten Ebene sind linksbündig angeordnet.

Heaps lassen sich in Arrays einbetten, dabei liegt die Wurzel in $a[0]$, das linke Kind von $a[i]$ in $a[2 * i + 1]$ und das rechte Kind in $a[2 * i + 2]$.
Die Höhe eines Heaps in Abhängigkeit von seinen Elementen ist gegeben durch $H = \lfloor log_2(n) \rfloor$. Es ist $H \in O(\log\, n)$.
Für die Anzahl der Knoten im linken Teilbaum l und die Knotenzahl im rechten Teilbaum r gilt stets: $r \leq l \leq 2r + 1$.
Ein Heap wird Reihe für Reihe von oben nach unten aufgebaut, indem neue Elemente möglichst weit links angefügt werden. Ist ein neues Element größer als sein Elternknoten, wird rekursiv nach oben getauscht.

4.6 Heapsort

Im Max-Heap enthält die Wurzel immer die größte Ziffer, deshalb tauschen wir die Wurzel mit dem letzten Element des Heaps, das in der untersten Ebene ganz rechts steht. Nun ist die letzte Position im Heap sortiert, wir können den Heap nun als Array aufschreiben und haben die letzte Position sortiert.
Dieses Verfahren wird rekursiv angewendet, indem das letzte, sortierte Element gestrichen wird. Anschließend muss wieder ein Max-Heap aufgebaut werden und wir beginnen von vorne, bis im Heap keine Elemente mehr enthalten sind, sodass das Array sortiert ist.

Auch Heapsort hat eine Laufzeitkomplexität von $O(n*log(n))$, der Speicherplatzbedarf ist konstant. Heapsort ist ein Inplace-Sortieralgorithmus und nicht stabil.

4.7 Prioritätswarteschlange

Elemente werden mit Schlüsseln versehen, wobei jeder Schlüssel höchstens an ein Element vergeben wird. Die Schlüssel betrachten wir nun als Priorität. Prioritätswarteschlangen können mit einem Array, einem sortierten Array oder einem Heap implementiert werden.

4.8 Quicksort

Wir wählen ein beliebigies Pivotelement und teilen das Array danach in zwei Teile - der linke Teil enthält Elemente, die kleiner als das Pivotelement sind, der rechte Teil enthält Elemente, die mindestens so groß sind wie das Pivotelement, allerdings nicht das Pivotelement selbst. Das wiederholen wir rekursiv. Die Pivotelemente sind am Ende in korrekter Reihenfolge.

Die Platzkomplexität von Quicksort ist $\Theta(log(n))$, es ist $W(n) \in \Theta(n^2)$, $B(n) \in O(n * log(n))$ und $A(n) \in O(n * log(n))$.

4.9 Bubblesort

Es werden alle nebeneinanderstehenden Paare in der Liste vertauscht, die momentan in der falschen Ordnung stehen. Die Liste rechts von der letzten Tauschposition des letzten Durchgangs ist dabei stets bereits sortiert. Sobald keine Änderungen mehr auftreten, ist der Algorithmus fertig - ansonsten muss der Algorithmus erneut durchlaufen.

Damit ist $W(n) \in O(n^2)$.

4.10 Countingsort

Die Elemente des Array sind aus dem Bereich $\{0, ..., B\}$ für einen bekannten Wert B. Wir erstellen ein Counting-Array, das so viele Positionen wie annehmbare Werte hat - jedem annehmbaren Wert wird (fortlaufend) eine Position im Counting-Array zugeordnet. Dieses Array wird mit Nullen initialisiert.
Nun gehen wir das zu sortierende Array durch. Für jedes Element im Array wird der zum Element gehörende Wert im Counting-Wert um 1 inkrementiert. Nun ersetzen wir im Counting-Array die Elemente durch die jeweiligen Teilsummen ($c[0] = c[0]$, $c[1] = c[0] + c[1]$, ...).
Abschließend gehen wir das unsortierte Array von hinten nach vorne durch. Bei jedem Element des unsortierten Arrays steht nun die neue, sortierte Position des Elements im Counting-Array. Bei jedem Einfügen eines Elements muss der zu diesem Element gehörige Wert im Counting-Array um 1 dekrementiert werden.
Die Laufzeit von Countingsort ist $\in O(n + B)$, der Speicherbedarf $\in O(n + B)$.

5 Elementare Datenstrukturen

5.1 Abstrakter Datentyp (ADT)

Ein abstrakter Datentyp besteht aus einer Datenstruktur (Menge von Werten) und einer Menge von Operationen darauf, zum Beispiel aus Konstruktor, Zugriffsfunktionen und Bearbeitungsfunktionen. Damit sind die Daten außerhalb des ADT nur über wohldefinierte Operationen zugänglich (Datenkapselung), während die Repräsentation der Daten nur für die Implementierung des ADT relevant ist.
Es gibt zum Beispiel *Stack*, *Queue*, *Priority-Queue* und einfach bzw. doppelt verkettete Listen.

5.2 Binärbaum

Binärbäume enthalten Knoten mit einem linken und einem rechten Nachfolger (die jeweils leer sein können). Es gibt eine Wurzel (ganz oben), innere Knoten mit mindestens einem Nachfolger und Blätter ohne Nachfolger.
Die Tiefe eines Knotens ist der Abstand des Knoten zur Wurzel, die Höhe eines Baumes ist die (maximale) Tiefe des untersten Knotens. Eine Ebene d enthält maximal 2^d Knoten, ein Binärbaum der Höhe h enthält höchstens $2^{h+1} - 1$ Knoten. Ein Binärbaum ist vollständig, wenn die möglichen Knotenzahl für die Höhe des Baumes voll ausgefüllt ist.
Bei einer Traversierung werden alle Elemente des Baumes einmal besucht, jede Kante wird dabei zweimal durchlaufen. Die Traversierung beginnt und endet an der Wurzel des Baumes. Die Teilbäume der Knoten werden in einer festen Reihenfolge (zuerst linker, dann rechter Teilbaum) besucht.
Es gibt verscheidene Traversierungen, die sich alleine dadurch unterscheiden, wann sie die Elemente des Knotens „besuchen“. So gibt es Inorder-, Preorder- und Postorder-Traversierung.

5.3 Binäre Suchbäume

Ein binärer Suchbaum (BST) ist ein Binärbaum, der Elemente mit Schlüsseln enthält, wobei der Schlüssel jedes Knotens mindestens so groß ist wie jeder Schlüssel im linken Teilbaum und höchstens so groß ist wie jeder Schlüssel im rechten Teilbaum.
Im binären Suchbaum gibt es neben dem Element des Knotens und einem Verweis auf die Nachfolger auch noch einen Verweis auf den Vaterknoten - dieser ist bei der Wurzel leer.
Die Inorder-Traversierung gibt aufgrund des Aufbaus des Suchbaums alle Schlüssel des Baumes in sortierter Reihenfolge aus.
Kompliziert ist im BST vor allem das Löschen, wenn das zu löschende Element in einem Knoten mit zwei Kindern steht (bei einem Kind wird der Knoten entfernt und der Zeiger vom Vaterknoten auf das Kind gelegt, ein Blatt wird einfach entfernt). Bei zwei Kindern finden wir den Nachfolger des zu löschenden Knotens, entfernen ihn aus seiner ursprünglichen Position und ersetzen den zu löschenden Knoten mit ihm.
Alle Operationen auf einem BST sind linear und $\in \Theta(h)$.

5.4 Rotationen in Bäumen

Bei einer Linksrotation rückt der rechte Nachfolger (N) des Knotens (K) an die Position von K. K wird linker Nachfolger von N, der ehemalige linke Nachfolger von N wird rechter Nachfolger von K. Die Rechtsrotation verläuft analog.
Durch die Rotation bleibt ein BST weiterhin ein BST, auch die Inorder-Traversierung bleibt unverändert. Die Zeitkomplexität der Rotation ist $\in \Theta(1)$.

5.5 AVL-Bäume

Ein AVL-Baum ist ein balanzierter BST, bei dem sich für jeden Knoten die Höhen der beiden Teilbäume um höchstens 1 unterscheiden - die Höhe der Teilbäume wird balanziert. Es gibt ein zusätzliches Datenfeld für jeden Knoten, in dem über die Höhe des Unterbaums Buch geführt wird.

Nach jeder Operation muss die Bilanz des AVL-Baums wiederhergestellt werden, dies ist in Θh möglich. Dadurch ist stets $h \in \Theta log(n)$ gewährleistet, $\Theta log(n)$ kann für alle Operationen auf dem Baum garantiert werden.
Es kann - wie bereits angerissen - passieren, dass der AVL-Baum nach dem Einfügen eines Knotens unbalanziert ist und die Bedingungen damit verletzt sind. Dann muss der Baum neu balanziert werden. Dafür gibt es folgende Regeln:

- Wenn das neue Blatt außen am Baum hängt, reicht eine einfache Rotation nach innen um den Vaterknoten.
- Wenn das neue Blatt innen am Baum hängt, muss eine Doppelrotation durchgeführt werden. Dazu rotieren wir erst um den Vaterknoten nach außen, sodass sich dann eine Situation wie im ersten Fall ergibt. Es wird nun um den neuen Vaterknoten nach innen rotiert.

Auch durch Löschen (von Blättern) kann der AVL-Baum unbalanziert werden, dann muss ebenfalls durch Rotieren Abhilfe geschaffen werden.

5.6 Rot-Schwarz-Bäume (RBT)

In Rot-Schwarz-Bäumen gibt es zwei Arten von Knoten: Schwarze Knoten werden strikt balanziert, während rote Knoten nur als „Schlupf“ dienen. Dabei muss die Anzahl der Schlupfknoten beschränkt bleiben. Die Balanzierungsanforderungen an die AVL-Bäume werden damit aufgeweicht, außerdem reduziert sich der benötigte Speicherplatz für zusätzliche Informationen auf 1 Bit (die Farbe), da der Balanzierungsfaktor nicht mehr gespeichert wird.
Die Wurzel eines RBT ist immer schwarz - jeder Knoten muss entweder rot oder schwarz sein. Außerdem darf jeder rote Knoten nur schwarze Kinder haben. Externe Knoten (Null-Zeiger, also quasi nicht existente Knoten) sind immer schwarz. Für jeden Knoten müssen alle Pfade, die an diesem Knoten starten und in einem externen Knoten enden, die gleiche Anzahl schwarzer Knoten enthalten. In Zeichnungen werden die externen Knoten häufig weggelassen.
Die Schwarzhöhe $bh(x)$ eines Knotens x ist die Anzahl schwarzer Knoten bis zu einem (externen) Blatt (x zählt nicht mit, das Blatt/der externe Knoten schon), die Schwarzhöhe $bh(t)$ eines RBT t ist die Schwarzhöhe seiner Wurzel.
Ein Rot-Schwarz-Baum t mit der Schwarzhöhe $h = bh(t)$ mindestens $2^h - 1$ und höchstens $4^h - 1$ innere Knoten - das sind alle „internen“ Knoten.
Neue Knoten werden immer als rote Knoten eingefügt, weil das Einfügen von schwarzen Knoten im Regelfall zur Verletzung der Schwarz-Höhen-Bedingung führt. Die Position für das Einfügen suchen wir uns wie in einem BST. Allerdings kann durch das Einfügen eines roten Knotens eine Rot-Rot-Verletzung auftreten, wenn bereits der Vater des neu einzufügenden Knotens rot ist:

1. Wenn der Onkel des Vaters ebenfalls rot ist, können wir den Vater und den Onkel schwarz sowie den Großvater rot färben, um die Rot-Rot-Verletzung zu beheben. Eventuell entstehen dadurch überhalb des Vaters erneut Rot-Rot-Verletzungen die iterativ genauso behoben werden müssen. Sofern die Wurzel durch der Vertauschung rot gefärbt wurde, können wir diese einfach wieder schwarz färben - die Schwarz-Höhe des Baumes erhöht sich dadurch um eins, eine Schwarz-Höhen-Verletzung tritt dadurch aber nicht auf.

2. Wenn der Onkel des Vaters nicht rot ist und das neue Blatt im Baum innen hängt, müssen wir über den Vater nach außen rotieren. Dann haben wir Fall 3.

3. Wenn der Onkel des Vaters nicht rot ist und das neue Blatt im Baum außen hängt, muss zunächst nach rechts um den Großvater rotiert werden. Sodann wird der ehemalige Großvater rot gefärbt, damit die Schwarz-Höhe von $(x + 1)$ wieder auf x korrigiert wird. Der neue Vater wird dann schwarz gefärbt.

Auch das Löschen von schwarzen Knoten in einem RBT ist problematisch, weil dadurch in der Regel eine Schwarz-Höhen-Verletzung entsteht - hingegen ist das Löschen von roten Knoten unproblematisch. Auch hier gibt es mehrere Fälle:

1. Hat der zu löschende Knoten zwei Kinder, suchen wir seinen Nachfolger und entfernen ihn aus dem Rot-Schwarz-Baum. Eine eventuell neu auftretende Farbverletzung ist dabei zu beheben. Nun ersetzen wir den zu löschenden Knoten durch seinen Nachfolger und übernehmen die Farbe des Nachfolgers.

2. Hat der zu löschende Knoten nur ein Kind, kann der Knoten gelöscht werden. Anschließend kann die Farbverletzung durch einfaches Vertauschen der Farben oder die „Weitergabe“ des Schwarzwertes Richtung Wurzel behoben werden.

3. Hat der zu löschende Knoten keine Kinder, muss der Onkel und der Vater unter Beachtung der Schwarzhöhe umgefärbt werden. Die eventuell dadurch entstehenden Farbverletzungen müssen iterativ behoben werden.

Die Laufzeit der Algorithmen für Einfügen und Löschen liegt in $\Theta(log(n))$, alle anderen Operationen sind mit denen in einem BST algorithmisch und bezüglich der Laufzeit identisch.

6 Hashing

6.1 Dictionary

Ein *Dictonary* (auch *assoziatives Array)* speichert Daten, die jederzeit anhand ihres Schlüssels abgerufen werden können. Die Daten sind dabei dynamisch gespeichert.

6.2 Direkte Adressierung

Bei der direkten Adressierung gibt es ein Array (die Direkte-Adressierungs-Tabelle) so, dass es für jeden möglichen Schlüssel eine entsprechende Position gibt. Jedes Array-Element enthält dabei einen Pointer auf die gespeicherte Information.

6.3 Grundlagen des Hashing

Ein Hash ist ein Algorithmus, der aus einer großen Datenmenge eine sehr kleine Zusammenfassung (Identifikation; Fingerabdruck) berechnet. Dabei ist er klar definiert und einfach sowie ressourcenschonend zu berechnen.

Das Ziel von Hashing ist es, einen extrem großen Schlüsselraum auf einen vernünftigen, kleinen Bereich von ganzen Zahlen abzubilden. Dabei soll es möglichst unwahrscheinlich sein, dass zwei Schlüssel auf die gleiche Zahl abgebildet werden. In der Praxis wird hierbei oft nur ein kleiner Teil der Schlüssel verwendet, damit ist bei der direkten Adressierung ein Großteil des Arrays verschwendet. $h(k)$ sei der Hashwert von k. Das Auftreten von $h(k) = h(k')$ für $k \neq k'$ nennt man Kollision.

6.4 Hashfunktionen

Eine gute Hash-Funktion ist einfach zu berechnen, surjektiv auf der Zielmenge und verwendet alle Indizes mit möglichst gleicher Häufigkeit. Ferner sollen ähnliche Schlüssel möglichst breit auf die Hashtabelle verteilt werden. Es gibt einige Basistechniken, um eine gute Hashfunktion zu erhalten:

Bei der Divisionsmethode wird die Hashfunktion mit $h(k) = k \bmod m$ gebildet. Dabei muss man m sorgfältig wählen, da zum Beispiel bei $m = 2^p$ einfach die letzten p Bits genommen werden. Eine gute Wahl ist m prim und nicht zu nah an einer Zweierpotenz.

Die Multiplikationsmethode stellt $h(k) = \lfloor m * (k * c \bmod 1) \rfloor$ mit $0 < c < 1$. Der Parameter m ist dabei nicht kritisch, Knuth empfiehlt $c = \frac{(\sqrt{5}-1)}{2} \approx 0,618$.

6.5 Universelles Hashing

Das größte Problem beim Hashing ist, dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den gleichen Hashwert abgebildet werden. Die Idee zur Lösung dieses Problem ist, zufällig eine Hashfunktionen aus einer gegebenen, kleinen Menge H auszuwählen - unabhängig von den verwendeten Schlüsseln.

6.6 MD5

MD5 übersetzt Nachrichten beliebiger Länge in einen 128-Bit-Hashwert. Inzwischen gilt MD5 als sehr unsicher, Kollisionen sind verhältnismäßig einfach zu berechnen.

6.7 Verkettung

Bei der Verkettung wird in der Hash-Tabelle an der Stelle des gehashten Schlüssels nicht nur der Wert gespeichert, sondern auch der Original-Schlüssel (vor dem Hashen). Die Speicherung erfolgt in der Hash-Tabelle als verkettete Liste. Dadurch erhöht sich natürlich den Speicheraufwand, außerdem muss beim Speichern und Abrufen eine Suche durchgeführt werden.

6.8 Offene Adressierung

Bei der offenen Adressierung werden alle Schlüssel direkt an einer Position in der Hashtabelle gespeichert (im Gegensatz zur Verkettung). Wenn ein Schlüssel eingefügt werden soll, wird geprüft, ob die entsprechende Position in der Hashtabelle frei ist. Ist dies nicht der Fall, kann der Schlüssel

nicht (wie bei der Verkettung) einfach angehangen werden. Es muss dann - in Abhängigkeit des einzufügenden Schlüssels - nach einer freien Position gesucht werden.
Damit das funktioniert, bekommt die Hashfunktion $h(k)$ nun einen zweiten Parameter, der in $[0, m-1]$ (m ist die Größe der Hashtabelle) liegt. Dieser Parameter gibt die Nummer der Sondierung (das ist die Nummer des Versuchs, eine freie Position zu finden) an. Die Hashfunktion liefert nun in der Regel eine andere Position, die wir in der Hashtabelle prüfen. Sobald wir eine leere Position finden, wird der Schlüssel dort in der Hashtabelle eingefügt und wir sind fertig. Finden wir keine leere Position, führen wir das Verfahren iterativ durch, erhöhen also den zweiten Parameter der Hashhfunktion in jedem Durchlauf.
Über den eindeutigen Index des Schlüssels in der Hashtabelle kann man dann mithilfe einer Direkten-Adressierungs-Tabelle auf das eigentlich assoziativ zu speichernde Element verweisen.
Die Suche in der Hashtabelle nach einem Schlüssel verläuft ähnlich - wir gehen wieder nach dem Schema durch die Tabelle und sind fertig, sobald wird den gesuchten Schlüssel gefunden haben. Es wird also nicht geprüft, ob die jeweilige Position leer ist, sondern ob der Inhalt der Hashtabelle an dieser Position dem gesuchten Schlüssel entspricht.
Aufpassen müssen wir beim Löschen eines Elements. Man könnte einfach wieder die Position des Elements suchen und dann die Hashtabelle an dieser Stelle leeren. Das führt aber zu dem Problem, dass alle eventuell nach diesem Element eingefügte Elemente nicht mehr gefunden werden, da die Sondierung früher abgebrochen wird. Daher ersetzen wir das Element beim Löschen durch z.B. "DELETED". Dieser Wert wird bei der Sondierung als leer interpretiert - beim Einfügen wird der Slot wiederverwendet, bei der Suche übersprungen.
Dadurch sind die Suchzeiten nicht mehr allein vom Füllgrad der Hashtabelle abhängig, deswegen verwendet man zur Kollisionsauflösung meist Verkettung, wenn Schlüssel auch gelöscht werden sollen.

6.9 Sondierung

Wir gehen nun näher auf das Verfahren der Sondierung ein. Für jeden Schlüssel k benötigen wir eine Sondierungssequenz $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$, um die offene Adressierung nutzen zu können. Dabei sollte die Sondierungssequenz eine Permutation von $\langle 0, ..., m-1\rangle$ sein, damit jeder Slot auch wirklich verwendet wird. Ideal wäre gleichverteiltes Hashing, sodass jede der $m!$ Permutationen als Sondierungssequenz gleich wahrscheinlich ist. In der Praxis ist das aber zu aufwändig, daher wird approximiert.

Lineares Sondieren
$h(k,i) = (h'(k) + i)$ mod m ist eine Hashfunktion für lineares Sondieren. Dabei ist h' eine normale Hashfunktion (siehe 6.4). Die Verschiebung der nachfolgenden Sondierungen hängt damit linear von i ab. Damit wird die gesamte Sequenz bereits durch die erste Sondierung bestimmt und es gibt nur m verschiedene Sequenzen.
Das führt zum Problem des Clusterings: Lange Folgen von belegten Slots tendieren dazu, noch länger zu werden, weil $h'(k)$ konstant bleibt und nur der Offset jedes Mal um eins größer wird.

Quadratisches Sondieren
$h(k,i) = (h'(k)+c_1*i+c_2*i^2)$ mod m ist eine Hashfunktion für quadratisches Sondieren. Dabei ist h'

eine normale Hashfunktion (siehe 6.4) und $c_1, c_2 \in \mathbb{N}$ sind geeignete Konstanten. Die Verschiebung der nachfolgenden Sondierungen hängt damit quadratisch von i ab. Damit wird die gesamte Sequenz auch bei der quadratischen Sondierung bereits durch die erste Sondierung bestimmt und es gibt nur m verschiedene Sequenzen.
Das Problem des Clusterings wird vermieden oder zumindest abgeschwächt. Allerdings tritt immer noch sekundäres Clustering auf, denn $h(k,0) = h(k',0)$ verursacht $h(k,i) = h(k',i)$ für alle i.

Doppeltes Hashing
$h(k,i) = (h_1(k)+i*h_2(k))$ mod m ist eine Hashfunktion für doppeltes Hashing. Dabei sind h_1 und h_2 normale Hashfunktionen (siehe 6.4). Die Verschiebung der nachfolgenden Sondierungen hängt damit nur von h_2 ab. Damit wird die gesamte Sequenz nicht bereits durch die erste Sondierung bestimmt. Deswegen werden die Schlüssel besser in der Hashtabelle verteilt und es wird eine Approximierung von gleichverteiltem Hashing erzielt.
Sind h_2 und m relativ prim, wird die gesamte Hashtabelle abgesucht. Man kann zum Beispiel $m = 2^k$ wählen und h_2 nur ungerade Zahlen erzeugen lassen, um ein gutes Ergebnis zu erzielen. Dann können m^2 verschiedene Permutationen erzeugt werden, weil jedes mögliche Paar von $h_1(k)$ und $h_2(k)$ eine andere Sequenz erzeugt.

7 Graphalgorithmen

7.1 Graphen

Graphen haben wir in der Vorlesung *Diskrete Strukturen* definiert, diese bestehen aus dem Paar (V,E) (Ecken/Knoten und Kanten). Die Kanten können gerichtet sein (eine Richtung haben) oder nicht - wir haben dann einen gerichteten bzw. ungerichteten Graphen.
Der Knoten u ist adjazent zu Knoten v gdw. $(u,v) \in E$ und inzident zu der Kante $e = (u,v)$. Der Grad $deg(u)$ ist die Anzahl der zu u inzidenten Kanten. Wir definieren - insbesondere für gerichtete Graphen - ferner den In-Grad $deg^+(u)$ und den Aus-Grad $deg^-(u)$. Ein Graph ist k-regulär, wenn alle Knoten den Grad k haben.
In ungerichteten Graphen $G = (V,E)$ gilt das Handshake-Lemma $\sum_{v \in V} deg(v) = 2 * |E|$, bei gerichteten Graphen ist $\sum_{v \in V} deg^+(v) = \sum_{v \in V} deg^-(v)$.
Ein Teilgraph eines Graphen $G = (V,E)$ ist ein Graph $G' = (V',E')$ mit $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$. Ist $V' \subset V$ und/oder $E' \subset E$, so nennen wir G' einen echten Teilgraphen von G. Wenn $E' = E \cap (V' \times V')$ ist, so ist G' der durch V' induzierte Teilgraph.
Ein gerichteter Graph G heißt symmetrisch, wenn aus $(v,w) \in E$ immer auch $(w,v) \in E$ folgt. Zu jedem ungerichteten Graphen gibt es einen korrespondierenden symmetrischen, gerichteten Graphen.
Der Graph G heißt vollständig, wenn jedes Paar von Knoten mit einer Kante verbunden ist. Es ist K_n der ungerichtete, vollständige Graph mit n Knoten.
Ferner können wir einen Graphen **transponieren**, der transponierte Graph von $G = (V,E)$ ist gegeben durch $G^T = (V,E')$ mit $(v,w) \in E'$ gdw. $(w,v) \in E$.

7.2 Pfade und Kreise

Ein Spaziergang (walk) von einem Knoten v zu einem Knoten w ist eine Folge $v_0, v_1, ..., v_k$ mit dem Startknoten $v_0 = v$ und dem Endknoten $v_k = w$. Ferner muss (v_i, v_{i+1}) für $0 \leq i \leq k-1$ eine Kante des Graphen sein.
Hat der Spaziergang nur paarweise verschiedene Knoten (d.h. $v_i \neq v_j$ für $i \neq j$), nennt man ihn simpel/einfach oder Pfad. Die Länge eines Spaziergangs ist die Anzahl der durchlaufenen Kanten.
Ein Kreis ist eine Spezialisierung des Pfades, dieser darf nicht-leer sein und es muss $v_0 = v_k$ sein - der Spaziergang muss also an der Stelle wieder enden, an der er begonnen hat, wir machen also eine Rundwanderung. Ein Graph ist kreisfrei oder azyklisch, wenn er keine Kreise als Teilgraphen hat.
Ein Knoten w ist vom Knoten v aus erreichbar, wenn es einen Pfad von v nach w gibt. Ein ungerichteter Graph G ist zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist. Eine Zusammenhangskomponente (connected component) von G ist ein maximaler zusammenhängender Teilgraph von G.
In einem gerichteten Graphen G' differenzieren wir genauer. G' ist stark zusammenhängend (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist. Hingegen ist G' schwach zusammenhängend, wenn der zugrunde liegende ungerichtete Graph (in dem alle Kanten ungerichtet gemacht worden sind) zusammenhängend ist. Eine starke Zusammenhangskomponente von G' ist ein maximaler stark zusammenhängender Teilgraph von G'.
Jeder ungerichtete Graph kann eindeutig in Zusammenhangskomponenten, jeder gerichtete Graph eindeutig in starke Zusammenhangskomponenten aufgeteilt werden.
Ein Baum ist ein kreisfreier und zusammenhängender Graph. Hat ein Baum n Knoten, so hat er $n-1$ Kanten. Ein ungerichteter Graph mit n Knoten und $n-2$ oder weniger Kanten kann nicht zusammenhängend sein, während ein ungerichteter Graph mit n Knoten und n oder mehr Kanten einen Zyklus enthalten muss - in beiden Fällen kann dieser Graph also kein Baum sein.

7.3 Repräsentation von Graphen

Es stellt sich die Frage, wie man einen Graphen als Datenstruktur am sinnvollsten repräsentiert. Hierzu gibt es verschiedene Möglichkeiten, wir machen uns die Adjazenz zu Nutzen.

Adjazenzmatrix
Wir haben einen Graphen $G = (V, E)$ mit $|V| = n$, $|E| = m$ und $V = v_1, ..., v_n$. Dann ist der Graph in der Adjazenzmatrix-Darstellung gegeben durch eine $n \times n$-Matrix A mit $A(i, j) = 1$ wenn $(v_i, v_j) \in E$ (und 0 wenn nicht). Wenn G ungerichtet ist, ist A symmetrisch ($A^{tr} = A$), dann muss nur die Hälfte der Matrix gespeichert werden. Der Platzbedarf ist $\Theta(n^2)$.

Adjazenzliste
Wir können einen Graphen auch als Array von Adjazenzlisten darstellen. Dabei ist die Arrayposition gegeben durch die Nummer des Knotens i, an dieser Stelle enthält das Array eine verkettete Liste mit allen Kanten mit Startknoten v_i. Wenn G ungerichtet ist, wird jede Kante zweimal gespeichert. Der Platzbedarf ist $\Theta(n + m)$.

7.4 Graphendurchlauf

Viele Algorithmen untersuchen jeden einzelnen Knoten und jede einzelne Kante eines Graphen. Dazu gibt es verschiedene Graphendurchlaufstrategien, die jeden Knoten (oder jede Kante) genau einmal besuchen: Die Breitensuche (BFS) und die Tiefensuche (DFS). Dabei handelt es sich im Wesentlichen um Verallgemeinerungen der Strategien zur Baumtraversierung.
Allerdings müssen wir uns alle bereits besuchten Knoten explizit merken. Wir arbeiten mit der Adjazenzlisten-Darstellung, Algorithmen auf dieser Basis kosten $O(|V| + |E|)$ Zeit.

Breitensuche
Bei der Breitensuche (Breadth-First Search, BFS) sind anfangs alle Knoten als unbesucht (WHITE) markiert. Es wird in einem beliebigen Knoten v begonnen. Der aktuelle Knoten wird immer als aktiv (GRAY) markiert. Sodann wird für jede Kante (v, w) mit unbesuchtem Nachfolger w von allen Kanten dieses Nachfolgers aus weitergesucht. Insbesondere gibt es keinerlei Backtracking. Der Knoten v wird dann als besucht (BLACK) markiert. Damit sind die schwarzen Knoten dann genau jene Knoten, die vom Startknoten aus erreichbar sind.
Für einen Knoten $v \in V$ bezeichnet $d(v)$ den Abstand vom Startknoten zum Knoten v, also die Anzahl der Kanten auf dem kürzesten Weg vom Startknoten nach v. Wenn ein Knoten w in die Queue gegeben wird, so färben wir die dafür verantwortliche Kante (v, w) gelb. Dann ist v der Vater vom Knoten w.
BFS besucht die Knoten also in einer Reihenfolge mit ansteigendem Abstand vom Startknoten. Erst wenn alle Knoten mit Abstand d verarbeitet sind, werden die Knoten mit Abstand $d+1$ angegangen. Die Suche terminiert, wenn für einen Abstand d keine Knoten auftreten. Die zu verarbeitenden Knoten werden als FIFO-Queue organisiert, es gibt nur eine einzige Verarbeitungsmöglichkeit für v (wenn es aus der Queue entnommen wird). Die gelben Kanten induzieren den Breitensuchbaum, der den Startknoten als Wurzel hat.
Die Zeitkomplexität der BFS ist $O(|V| + |E|)$, der Platzbedarf ist $\Theta(|V|)$.

Tiefensuche
Auch bei der Tiefensuche (Depth-First Search, DFS) markieren wir zunächst alle Knoten als unbesucht (WHITE). Der aktuelle Knoten v wird immer als aktiv (GRAY) markiert, es ist beliebig, mit welchem Knoten wir beginnen. Für jede Kante (v, w) mit unbesuchtem Nachfolger w suchen wir nun rekursiv von w aus. Das bedeutet also, dass wir den neu entdeckten Knoten w besuchen und von dort aus forschen, bis es nicht mehr weiter geht. Sodann gehen wir von w nach v zurück. Für jede Kante (v, w), für die der Nachfolger w schon besucht wurde, überprüfen wir die Kante, besuchen w aber nicht. Abschließend markieren wir den Knoten v als besucht (BLACK). Damit sind auch bei der DFS die schwarzen Knoten diejenigen, die vom Startknoten aus erreichbar sind.

Die Tiefensuche erforscht also einen Pfad so weit wie möglich und setzt dann zurück. Die zu verarbeitenden Knoten werden also in LIFO-Reihenfolge verarbeitet. Es gibt zwei mögliche Verarbeitungszeitpunkte für jeden Knoten - wenn dieser entdeckt oder wenn er abgeschlossen wird.

Bei der Tiefensuche gibt es verschiedene Klassen von Kanten. Baum-Kanten treten im DFS-Baum auf. Rückwärts-Kanten gehen von Knoten u zum Vorfahren v, Vorwärts-Kanten gehen von einem Vorfahren v zu einem Knoten u. Alle restlichen Kanten sind Quer-Kanten. In einem ungerichteten

Graphen ist jede Kante entweder Baum-Kante oder Rückwärts-Kante.

Die Zeitkomplexität der DFS ist $O(|V| + |E|)$, der Platzbedarf ist $\Theta(|V|)$.

7.5 Finden von Zusammenhangskomponenten

TODO

7.6 VL15

TODO

8 Minimale Spannbäume

VL16

9 Kürzeste Pfade

VL17

10 Matching

VL18

11 Maximaler Fluss

VL19

12 Dynamische Programmierung

VL20

13 Greedy-Algorithmen

VL21

14 Algorithmische Geometrie

VL22