Lecture Summary

Note $\log n \equiv \log_2 n$

Note An einigen Stellen ist Wikipedia zitiert, oft weil die Erklärung einfach elegant ist.

Note This is not a strong subject of mine, that's why a) don't believe everything in this document and b) I copied a lot of stuff together but I tried to still make it useful.

Note Use freaking Google, there's plenty of video tutorials (from India) on many algorithms and, since this is a 101 course, there's many universities offering many lectures for free, e.g. the MIT (via OCW[1])!

Table of Contents

1 Einführung & Algorithmenentwurf 4
1.1 Einführung und Einheitskostenmodell 4
1.2 Algorithmen 4
1.2.1 Star 4
1.2.2 Maximum Subarray Sum 4
1.2.3 Karatsuba 4
2 Suchen 5
2.1 Unsortiert 5
2.2 Sortiert 5
2.3 Median 5
2.3.1 Randomisiert 5
2.3.2 Nach Blum. $\mathcal{O}n$ 5
3 Hashing 5
3.1 Wahl der Hashfunktion 5
3.2 Perfektes und universelles Hashing 6
3.3 Hashverfahren mit Verkettung der Überläufer 6
3.4 Open Hashing 6
3.5 **XX** Dynamische Hashverfahren 6
4 Sortieren 6
4.1 Insertionsort 6
4.2 Selectionsort 7
4.3 Bubblesort 7
4.4 Mergesort 7
4.5 Quicksort 7
4.5.1 Randomized Quicksort 8
4.6 Radixsort 8
4.7 Heapsort 8
4.8 Odd-even transposition sort 9

[1] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/

5 Suchbäume 9
5.1 Skipliste 9
5.2 Natürlicher binärer Suchbaum 9
5.3 Traversieren 9
5.4 AVL Bäume 10
5.5 Selbstanordnung in linearen Listen 10
5.6 Kompetitive und amortisierte Analyse 10
5.7 Optimale Suchbäume (mit DP) 11
5.8 Splay Trees 11
6 Dynamische Programmierung 12
6.1 Longest common subsequence 13
6.2 Editierdistanz (Levenshtein-Distanz) 13
6.3 Matrixkettenmultiplikation 13
6.4 Matrixmultiplikation nach Strassen 13
6.5 Subset sum 13
6.6 Rucksackproblem 14
7 Backtracking, Branch-and-Bound 14
7.1 n-Queens 14
7.2 Branch-and-bound 15
8 Graphenalgorithmen 15
8.1 Reflexive und transitive Hülle 15
8.2 Topologisches Sortieren 15
8.3 Traversieren 16
8.3.1 DFS, $OE = Om$ 16
8.3.2 BFS, $OE = Om$ 16
8.4 Minimaler spannender Baum (MST) 17
8.4.1 Greedy 17
8.4.2 Kruskal (mit Union-Find) 17
8.4.3 Prim/Dijkstra 17
8.5 Union-Find 17
8.6 Fibonacci Heap 18
8.6.1 Struktur 18
8.6.2 Operationen auf Fibonacci Heaps 18
8.6.3 Analyse 19
8.7 Shortest Path Problem 19
8.7.1 Bellman-Ford 19
8.7.2 Dijkstra 20
8.8 Zunehmende-Wege-Algorithmus nach Ford-Fulkerson 20
8.9 Max-Flow-Min-Cut 20

8.10 Max Flow 21
8.10.1 $Onm2$ nach Edmonds und Karp 21
8.10.2 $Om2n$ nach Dinic 21
8.11 Matching in bipartiten Graphen. Satz von Hall 21
9 Geometrische Algorithmen 21
9.1 Konvexe Hülle 21
9.1.1 Jarvis (gift wrapping algorithm) 21
9.1.2 Graham 22
9.1.3 Linearer Scan 22
9.2 Schnitt orthogonaler Liniensegmente prüfen 22
9.3 Schnitte beliebig orientierter Liniensegmente 23
9.4 Schnitt achsenparalleler Rechtecke 23
9.5 Range tree 24
9.5.1 1-dimensionaler range tree 24
9.5.2 d-dimensionaler range tree 24
9.6 Segment tree 25
9.6.1 Struktur und Operationen 25
9.7 Interval tree 26
9.7.1 Struktur und Operationen 26
9.7.2 Vergleich mit ähnlichen Datenstrukturen 26
9.8 Prioritätssuchbaum 27
9.9 Geometrisches Divide-and-conquer: Paar nächster Nachbarn in Punktmenge 27
10 Externspeicher-Datenstrukturen. 27
10.1 Principle of Locality 27
10.2 B-Bäume 27
10.2.1 Struktur von B-Bäumen 27
10.2.2 Die elementaren Operationen auf B-Bäumen 28
11 Additional Wisdom 29

1 Einführung & Algorithmenentwurf

1.1 Einführung und Einheitskostenmodell

Ein Algorithmus ist ein strukturiertes Verfahren zur schrittweisen Lösung eines Problems. Algorithmen spielen in der Informatik ein zentrale Rolle und haben auch einen entsprechenden Stellenwert. Um die Effizienz eines Algorithmus zu beschreiben, kann entweder das Einheitskostenmodell, bei dem jede Operation gleich lange benötigt, unabhängig davon, wie schnell sie tatsächlich ist, oder das seltener benutzte logarithmische Kostenmodell verwendet werden. Es gelten folgende Notationen:

- $f \in \mathcal{O}(g)$: g ist eine asymptotische obere Schranke,
$$\mathcal{O}(g) \coloneqq \{f: \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0: f(n) \leq c \cdot g(n)\}$$
- $f \in \Omega(g)$: g ist eine asymptotische untere Schranke,
$$\Omega(g) \coloneqq \{f: \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0: f(n) \geq c \cdot g(n)\}$$
- $f \in \Theta(g)$: g ist eine asymptotische scharfe Schranke, sowohl $f \in \mathcal{O}(g)$ als auch $g \in, \mathcal{O}(f)$,
$$\Theta(g) \coloneqq \{f: \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}\ \forall n \geq n_0: c^{-1} \cdot g(n) \leq f(n) \leq c \cdot g(n)\}$$

In der $\mathcal{O}$-Notation: $0 < 1/n < 1 < \log\log\log n < \log\log n < \sqrt{\log n} < \log n < \log^2 n < \log^3 n < \sqrt{n} < n < n \cdot \log n < n^2 < n^2 \cdot \log n < n^3 < 2^n < n \cdot 2^n < 3^n < n! < n^n < 2^{2^n}$

1.2 Algorithmen

1.2.1 Star

Ein Star ist eine Person, den alle kennen, aber der Star kennt niemanden. Dies impliziert, es kann genau ein Star geben. Um diesen zu finden kann naiv eine $n \times n$-Matrix (mit 0 in der Diagonalen) ausgefüllt werden, wozu $n^2 - n$ Felder ausgefüllt werden müssen, was äusserst ineffizient ist. Eine bessere (und sogar optimale, d.h. prüfen und finden ist gleich schnell) Variante ist mittels Induktion, bei dem anfangs nur eine Person im Raum ist und schrittweise eine Person mehr hinzukommt, welche dann gefragt wird, ob sie die anderen kenne. Laufzeit: $F(n) = F(n-1) + 2 + 1 = F(n-1) + 3 = F(n-2) + 3 + 3 = \cdots = F(2) + (n-2) \cdot 3 = 3n - 4 \in \mathcal{O}(n)$.

1.2.2 Maximum Subarray Sum

"The maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers (containing at least one positive number) which has the largest sum."[2] In der **naiven** Version werden alle n^2 möglichen Subarrays betrachtet was zu $\Theta(n^3)$ führt.

Ein zweiter Ansatz geht davon aus, dass einige Intervalle bereits vollständige Teilintervalle beinhalten. Dazu wird bei gleichbleibendem $i = 1 \dots n$ und wachsendem $k = i \dots n$ die Summe nicht verändert, ausser i verändert sich. Dadurch werden sogenannte Präfixsummen gebildet welche in $\mathcal{O}(n)$ gespeichert und berechnet werden können. Durch das Durchlaufen beträgt die Laufzeit $\mathcal{O}(n^2)$.

Die dritte Version verwendet **divide-and-conquer** und geht davon aus, dass die jeweils linke und rechte Hälfte mit Induktion lösbar ist. Im divide Schritt wird das Array zweigeteilt, $\mathcal{O}(1)$, im conquer Schritt mit Induktion die linke und die rechte Lösung hergestellt, $T(n/2)$ pro Seite, sprich $2 \cdot T(n/2)$. Im merge Schritt kann die Lösung (ausgehend davon, dass sie bereits gefunden wurde), entweder ganz rechts (r) oder ganz links (l) liegen oder über die Mitte hinausgehen (m), wobei sie in diesem Fall nicht gefunden wurde. Dazu werden ab der Mitte Präfixsummenberechnet, wobei jeweils um 1 nach links, nach rechts, dann um 2 nach links, nach rechts, … geschaut wird. Für die Rekursion, die bis zu einer Intervalllänge von 1 läuft, wird die Zahl genommen, falls sie positiv ist, sonst 0. Dieser Algorithmus ist $\mathcal{O}(n \cdot \log n)$.

Eine vierte Version verwendet einen **linearen Scan** und läuft in $\mathcal{O}(n)$. Beim linearen Scan gibt es zwei Möglichkeiten, entweder trägt a_i nichts zur Lösung bei oder es trägt zur Lösung bei und ist Teil der Lösung, dadurch ist die beste Präfixsumme $[1, a_i]$.

1.2.3 Karatsuba

Die klassische Schulmultiplikation ist $\mathcal{O}(n^2)$, Karatusba ist $\mathcal{O}\left(n^{\log_2 3}\right)$.

$$x = x_1 \cdot \underbrace{B^m}_{\text{Basis}} + x_2, y = y_1 \cdot B^m + y_2, \qquad a \coloneqq x_1 \cdot y_1, b \coloneqq x_2 \cdot y_2, \qquad c \coloneqq xy = (x_1 + x_2)(y_1 + y_2) - a - b$$

[2] Wikipedia

2 Suchen

Um ein Schlüssel $k \in \mathbb{N}$ in einer linearen Liste zu finden, gibt es verschiedene Methoden.

2.1 Unsortiert

Falls die Liste/Array unsortiert ist, muss k mit jedem a_i verglichen werden, was zu $\Theta(n)$ führt, auch die Unterteilung in Gruppen ändert nichts daran.

2.2 Sortiert

Falls das Array sortiert ist, kann mittels binärer Suche k in $\mathcal{O}(\log n + 1)$ gefunden werden. Da bei einer **binären Suche** der Ablauf der Entscheidungen als Baum mit n Knoten dargestellt werden kann (und die Länge die Geschwindigkeit bestimmt), ist die Mindesthöhe des Baumes $\approx \log n$ und die maximale Knotenanzahl bei Höhe h beträgt $\lfloor \log_2 n \rfloor$, wodurch folgt, dass die binäre Suche optimal ist.

Bei einer **Interpolationssuche** wird die Position von k interpoliert, jedoch kann dies im worst case zu $\mathcal{O}(n)$ (statt, wie erwartet, $\mathcal{O}(\log \log n)$) führen.

2.3 Median

„Der Median einer Auflistung von Zahlenwerten ist derjenige Wert, welcher an der mittleren Stelle steht, wenn man die Werte der Grösse nach sortiert."[3]

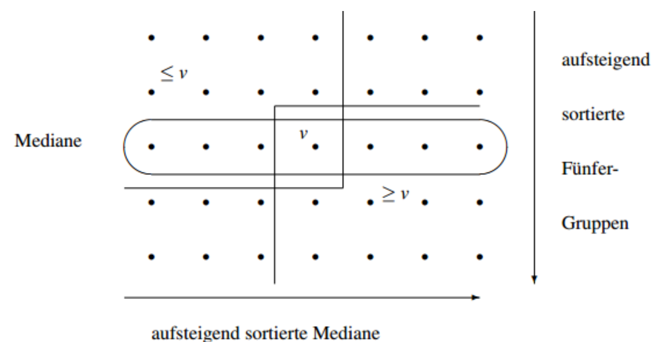
2.3.1 Randomisiert

Zuerst wird im Array ein Pivot p gewählt und nach diesem werden nun die a_i verschoben, grössere rechts, kleinere links, $\mathcal{O}(n)$.

2.3.2 Nach Blum. $\mathcal{O}(n)$

k-Algo

1. Bilde 5-er Gruppen von Zahlen, $\mathcal{O}(0)$
2. Je Gruppe, bestimme den Median, $\mathcal{O}(n)$
3. Wende *k-Algo* auf die Mediane (→ Mediane der Mediane) an, wobei dieser Median der Mediane als Pivot genommen wird
4. Aufteilen am Pivot, $\mathcal{O}(n)$
5. *K-Algo* auf Teil anwenden bis an der richtigen Position



3 Hashing

Sei $\mathcal{K} \subseteq \mathbb{N}_0$ das Schlüsseluniversum und folgende Operationen möglich: insert, delete, search. Je nach Datenstruktur benötigen diese Operationen unterschiedlich lange. In einem Array ist das Hashing dafür verantwortlich einem Element, basierend auf dessen Daten/Inhalt, eine Position zuzuweisen, wofür die Hashfunktion $h: \mathcal{K} \to \{0,1,\dots,n-1\}, \ |\mathcal{K}| \gg n$ zuständig ist. Diese Funktion, $h(k)$, sollte gut streuen, sprich die Schlüssel gut verteilen, und Kollisionen sollten effizient gelöst werden.

3.1 Wahl der Hashfunktion

Bei der **Divisions-Rest-Methode** ist $h(k) \coloneqq k \bmod m$, wobei die Wahl von m entscheidend ist. Für $m = 2^i$ wird ein Teil der Binärzahl abgeschnitten und nur die letzten i Stellen betrachtet, was an sich keine schlechte Variante ist. Für ein gerades m ist die Streuung sehr schlecht. Eine Wahl von m als Primzahl ist immer gut.

Bei der **multiplikativen Methode** wird der Schlüssel mit einer irrationalen Zahl multipliziert und dann wird der ganzzahlige Bereich abgeschnitten (irrationale Zahl $\in [0,1)$).

Beispiel[4]: $\phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.61803, \ h(k) = \left\lfloor m \underbrace{\left(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor\right)}_{\in[0,1)} \right\rfloor$

[3] Wikipedia

[4] Das Inverse des Goldenen Schnitts

3.2 Perfektes und universelles Hashing

Falls die einzufügenden Schlüssel im Vorfeld bekannt sind und $|k| \leq m$ ist eine kollisionsfreie Speicherung möglich. Eine Klasse von Hashfunktionen $\mathcal{H} \subseteq \{n: \mathcal{K} \to \{0, \dots, m-1\}\}$ heisst **universell**, wenn $\forall x \neq y: \delta(x, y, \mathcal{H}) \leq \frac{|\mathcal{H}|}{m}, \delta(x, y, h) := \begin{cases} 1, & \text{falls } h(x) = h(y) \text{ und } x \neq y \\ 0, & \text{sonst} \end{cases}$

3.3 Hashverfahren mit Verkettung der Überläufer

Bei einer **direkten** Verkettung werden die Elemente nicht in der Tabelle selbst sondern in einer an Liste an der Tabellenposition gespeichert. Dazu werden die Operationen search, insert und delete benötigt. Zusätzlich werden zum **Belegungsfaktor** α die Zeit c_n für die erfolgreiche und c'_n für die erfolglose Suche definiert. Bei diesem Verfahren ist $\alpha = c'_n = n/m$ und $c_n \approx 1 + \frac{\alpha}{2}$.

Bei der **separaten** Verkettung werden die kollidierenden Schlüssel „extern" gespeichert, $c'_n \approx \alpha + e^{-\alpha}, c_n \approx 1 + \frac{\alpha}{2}$.

Mit diesen Verfahren könne mehr Schlüssel, als die Tabelle Platz hat, gespeichert werden, jedoch gibt es einen grossen Overhead.

3.4 Open Hashing

$h(k) = s(j, k)$ für $j = 0,1, \dots, m-1$, bei der Suche wird der Eintrag an der Stelle $h(k) - s(j, k)$ betrachtet.

Das **lineare** Sondieren, $s(j, k) = j$ ist sehr einfach, hat aber auch einige Nachteile. $c_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$, $c'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$.

Um die Häufung etwas zu streuen werden beim **quadratischen** Sondieren, $j(j, k) = (\lceil j/2 \rceil)^2 \cdot (-1)^j$, die Schlüssel mehr gestreut.

Bei **Double Hashing** wird eine zweite Hashfunktion $h'(k)$ verwendet, wobei $s(j, k) = j \cdot h'(k)$. $c'_n \approx \frac{1}{1-\alpha}, c_n < 2.5$

3.5 XX Dynamische Hashverfahren

4 Sortieren

- http://www.sorting-algorithms.com/
- http://sorting.at/

Bei vergleichsbasierten Sortieralgorithmen (comparison sort) ist durch den Entscheidungsbaum, der $\geq n!$ Blätter hat (Anzahl der unterschiedlichen Permutationen), eine untere Schranke von $\mathcal{O}(n \cdot \log n)$ gegeben (was der Höhe des Baumes entspricht), sprich kein solcher Algorithmus kann schneller (z.B. $\mathcal{O}(n)$) sein.[5]

	STABIL	IN SITU	LAUFZEIT AVERAGE CASE (WORST)	VERGLEICHE MIN | MAX	VERTAUSCHUNGEN MIN | MAX
INSERTIONSORT	Ja	Ja	$\mathcal{O}(n^2)$	$\Theta(n) \| \Theta(n^2)$	$0 \| \Theta(n^2)$
SELECTIONSORT	Nein	Ja	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$0 \| \Theta(n)$
BUBBLESORT	Ja	Ja	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$0 \| \Theta(n^2)$
MERGESORT	(Ja)	Nein (Ja)	$\mathcal{O}(n \cdot \log n)$	N/A	N/A
QUICKSORT	Nein	Ja	$\mathcal{O}(n \cdot \log n)$ $(\mathcal{O}(n^2))$	N/A	N/A
HEAPSORT	Nein	Ja	$\mathcal{O}(n \cdot \log n)$	N/A	N/A

Prinzipiell sind die Erklärungen aus Wikipedia.

4.1 Insertionsort

„Das Vorgehen ist mit der Sortierung eines Spielkartenblatts vergleichbar. Am Anfang liegen die Karten des Blatts verdeckt auf dem Tisch. Die Karten werden nacheinander aufgedeckt und an der korrekten Position in das Blatt, das in der Hand gehalten wird, eingefügt. Um die Einfügestelle für eine neue Karte zu finden wird diese sukzessive (von links nach

[5] Siehe auch: Parallel Programming, Lecture 22, Slides 30 – 37 (nethz-Login benötigt)

rechts) mit den bereits einsortierten Karten des Blattes verglichen. Zu jedem Zeitpunkt sind die Karten in der Hand sortiert und bestehen aus den zuerst vom Tisch entnommenen Karten.“

4.2 Selectionsort

„Sei S der sortierte Teil des Arrays und U der unsortierte Teil. Am Anfang ist S noch leer, U entspricht dem ganzen Array. Das Sortieren durch Auswählen funktioniert so:

Suche das kleinste Element in U und vertausche es mit dem ersten Element.

Danach ist das Array bis zu dieser Position sortiert. Das kleinste Element wird in S verschoben. S ist um ein Element gewachsen, U um ein Element kürzer geworden. Anschliessend wird das Verfahren solange wiederholt, bis das gesamte Array abgearbeitet worden ist.“

4.3 Bubblesort

„In der Bubble-Phase wird die Eingabe-Liste von links nach rechts durchlaufen. Dabei wird in jedem Schritt das aktuelle Element mit dem rechten Nachbarn verglichen. Falls die beiden Elemente das Sortierkriterium verletzen, werden sie getauscht. Am Ende der Phase steht bei auf- bzw. absteigender Sortierung das grösste bzw. kleinste Element der Eingabe am Ende der Liste.

Die Bubble-Phase wird solange wiederholt, bis die Eingabeliste vollständig sortiert ist. Dabei muss das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da die restliche zu sortierende Eingabe keine grösseren bzw. kleineren Elemente mehr enthält.“

4.4 Mergesort

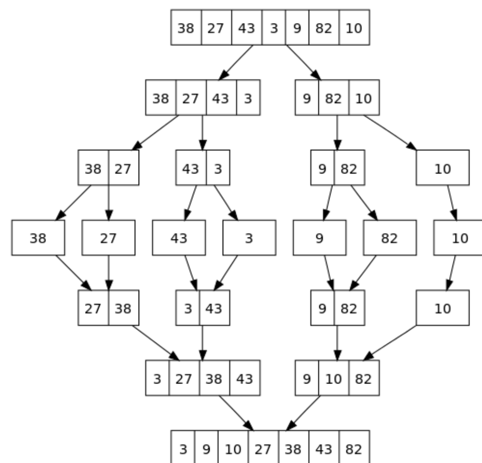


Bild links.

1. „Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.“

Mergesort ist sehr gut geeignet für z.B. GPGPU oder das Sortieren auf mehreren Datenträgern.

Beim **rekursiven** Mergesort werden die jeweiligen Hälften rekursiv bearbeitet. Bei der **iterativen** Version werden zwei verschachtelte Schleifen verwendet, um das Array durch zu iterieren.

„**Natural** Mergesort ist eine Erweiterung von Mergesort, die bereits vorsortierte Teilfolgen, so genannte runs, innerhalb der zu sortierenden Startliste ausnutzt. Die Basis für den Mergevorgang bilden hier nicht die rekursiv oder iterativ gewonnenen Zweiergruppen, sondern die in einem ersten Durchgang zu bestimmenden runs:

Diese Variante hat den Vorteil, dass sortierte Folgen „erkannt“ werden und die Komplexität im Best-Case $\mathcal{O}(n)$ beträgt. Average- und Worst-Case-Verhalten ändern sich hingegen nicht.“

4.5 Quicksort

„Zunächst wird die zu sortierende Liste in zwei Teillisten („linke“ und „rechte“ Teilliste) getrennt. Dazu wählt Quicksort ein sogenanntes Pivotelement aus der Liste aus. Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Teilliste, und alle, die grösser sind, in die rechte Teilliste. Die Elemente, die gleich dem Pivotelement sind, können sich beliebig auf die Teillisten verteilen. Nach der Aufteilung sind die Elemente der linken Liste kleiner oder gleich den Elementen der rechten Liste.

Anschliessend muss man also nur noch jede Teilliste in sich sortieren, um die Sortierung zu vollenden. Dazu wird der Quicksort-Algorithmus jeweils auf der linken und auf der rechten Teilliste ausgeführt. Jede Teilliste wird dann wieder in zwei Teillisten aufgeteilt und auf diese jeweils wieder der Quicksort-Algorithmus angewandt, und so fort. Wenn eine Teilliste der Länge eins oder null auftritt, so ist diese bereits sortiert und es erfolgt der Abbruch der Rekursion.

Das Verfahren muss sicherstellen, dass jede der Teillisten mindestens um eins kürzer ist als die Gesamtliste. Dann endet die Rekursion garantiert nach endlich vielen Schritten. Das kann z. B. dadurch erreicht werden, dass das ursprünglich als Pivot gewählte Element auf einen Platz zwischen den Teillisten gesetzt wird und somit zu keiner Teilliste gehört.“

Extra Speicherplatz: Linear, logarithmisch falls Rekursion im kurzen Teil, konstant

4.5.1 Randomized Quicksort[6]

Beim randomisierten Quicksort wird das Pivot, statt wie sonst üblich fix, z.B. ganz rechts, zufällig gewählt, wodurch ungünstige Eingaben kompensiert werden und die Laufzeit $\mathcal{O}(n \cdot \log n)$ beträgt.

$$T(n) = \sum_{k=1}^{n} T\underbrace{(k-1)}_{0\ldots n-1} + T\underbrace{(n-k)}_{0\ldots n-1} + n - 1 = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + n - 1 \leq 4n\log_2 n\,,\ T(1) = 0$$

$$T(n) \leq \frac{2}{n\sum_{k=1}^{n-1} 4k\log k + n - 1} = \frac{8}{n}\left(\sum_{k=1}^{n/2} k \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=n/2+1}^{n-1} k \underbrace{\log k}_{\leq \log n}\right) + n - 1$$

$$\leq \frac{8}{n}\left(\left(\sum_{k=1}^{n/2} k\right)(\log n - 1) + \sum_{k=n/2+1}^{n-1} k\log n\right) + n - 1 = \frac{8}{n}\left(\left(1 + \frac{n}{2}\right)\frac{n}{4}(\log n - 1) + \log\left(\frac{3n}{2}\right)\left(\frac{n}{2} - 1\right)\frac{1}{2}\right) + n - 1$$

$$= 2\log n + n\log n - 2 - n + 3n\log n - 6\log n + n - 1 = 4n\log n - 4\log n - 3 \leq 4n\log n \text{ q.e.d.}$$

4.6 Radixsort

„Bei Radixsort wird davon ausgegangen, dass die Schlüssel der zu sortierenden Daten nur aus Zeichen eines endlichen Alphabets bestehen. Zusätzlich muss eine Totalordnung zwischen den Zeichen des Alphabets bestehen.

Eine zweite Voraussetzung ist, dass die Länge ℓ der Schlüssel durch eine von vornherein bekannte Konstante begrenzt ist, da die Anzahl der Stellen pro Schlüssel eine entscheidende Auswirkung auf die Linearität des Laufzeitverhaltens hat.

Radixsort besteht aus zwei Phasen, die immer wieder abwechselnd durchgeführt werden. Die Partitionierungsphase dient dazu, die Daten auf Fächer aufzuteilen, während in der Sammelphase die Daten aus diesen Fächern wieder aufgesammelt werden. Beide Phasen werden für jede Stelle der zu sortierenden Schlüssel einmal durchgeführt.

Partitionierungsphase In dieser Phase werden die Daten in die vorhandenen Fächer aufgeteilt, wobei für jedes Zeichen des zugrundeliegenden Alphabets ein Fach zur Verfügung steht. In welches Fach der gerade betrachtete Schlüssel gelegt wird, wird durch das an der gerade betrachteten Stelle stehende Zeichen bestimmt.

Sammelphase Nach der Aufteilung der Daten in Fächer in Phase 1 werden die Daten wieder eingesammelt und auf einen Stapel gelegt. Hierbei wird so vorgegangen, dass zuerst alle Daten aus dem Fach mit der niedrigsten Wertigkeit eingesammelt werden, wobei die Reihenfolge der darin befindlichen Elemente nicht verändert werden darf. Danach werden die Elemente des nächsthöheren Faches eingesammelt und an die schon aufgesammelten Elemente angefügt. Dies führt man fort, bis alle Fächer wieder geleert wurden.

Diese beiden Phasen werden nun für jede Stelle der Schlüssel wiederholt, wobei mit der letzten Stelle begonnen wird und in der letzten Iteration die erste Stelle zum Aufteilen verwendet wird. Nach dem Aufsammeln für die erste Stelle der Schlüssel sind die Daten aufsteigend sortiert.“ $\mathcal{O}(n \cdot \ell)$

4.7 Heapsort

„Die Eingabe ist ein Array mit zu sortierenden Elementen. Als erstes wird die Eingabe in einen binären Max-Heap überführt. Aus der Heap-Eigenschaft folgt direkt, dass nun an der ersten Array-Position das grösste Element steht. Dieses wird mit dem letzten Array-Element vertauscht und die Heap-Array-Grösse um 1 verringert, ohne den Speicher freizugeben. Die neue Wurzel des Heaps kann die Heap-Eigenschaft verletzen. Die Heapify-Operation korrigiert gegebenenfalls den Heap, so dass nun das nächstgrössere bzw. gleich grosse Element an der ersten Array-Position steht. Die Vertausch-, Verkleiner- und Heapify-Schritte werden so lange wiederholt, bis die Heap-Grösse 1 ist. Danach enthält das Eingabe-Array die Elemente in aufsteigend sortierter Reihenfolge.“

Ein Heap ist ein voller Binärbaum mit Ausnahme der letzten Ebene, die von links her gefüllt wird, bei welchem immer das Maximum (≥) der Schlüssel in der Wurzel steht (dies gilt auch für alle Teilbäume). Durch die Struktur gegeben ist folgende Eigenschaft: ist die Wurzel das Element i, so sind dessen beiden Kinder $2i$ und $2i + 1$.

Wird ein Heap der Reihe nach (sprich von links nach rechts, von oben nach unten, $j = 1,2,3,\ldots,n-1,n$) traversiert, entspricht die Ausgabe der Repräsentation eines Heaps als Array.

„Zu Beginn muss natürlich erst ein solcher Heap aufgebaut werden, was sich in linearer Zeit erledigen lässt: Der Baum wird einfach mit den Elementen der Eingabe gefüllt und dann wird von unten her begonnen: Die

[6] Siehe auch http://www.cs.cmu.edu/~avrim/451f11/lectures/lect0906.pdf

Teilbäume auf der untersten Ebene sind bereits Heaps (da sie nur ein Element haben), und dann kann für jeden inneren Knoten die Prozedur versickern aufgerufen werden, damit die Heapbedingung überall erfüllt ist.“[7]

4.8 Odd-even transposition sort

“It functions by comparing all (odd, even)-indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for (even, odd)-indexed pairs (of adjacent elements). Then it alternates between (odd, even) and (even, odd) steps until the list is sorted.”

Dieser Algorithmus ist nicht ganz unähnlich zum bitonic sort, beide sind für parallele Ausführung geeignet.

5 Suchbäume

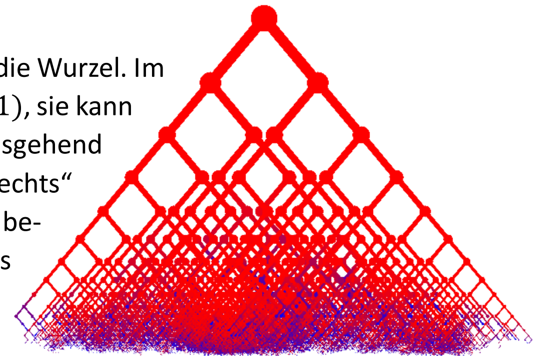
	SEARCH (AVG|WORST)	INSERT (AVG|WORST)	DELETE (AVG|WORST)
LINKED LIST	$\Theta(n)$	$\Theta(1)\|\Theta(n)$	$\Theta(n)$
SKIP LIST	$\mathcal{O}(\log n)\|\mathcal{O}(n)$	$\mathcal{O}(\log n)\|\mathcal{O}(n)$	$\mathcal{O}(\log n)\|\mathcal{O}(n)$
NATURAL BINARY SEARCH TREE	$\mathcal{O}(\log n)\|\mathcal{O}(n)$	$\mathcal{O}(\log n)\|\mathcal{O}(n)$	$\mathcal{O}(\log n)\|\mathcal{O}(n)$
AVL	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
SPLAY[8]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Die **Wörterbuchoperationen** sind: search, insert, delete.

5.1 Skipliste

„Wie verkettete Listen werden auch bei der Skipliste die Daten in Containern abgelegt. Diese enthalten einen Schlüssel und einen Zeiger auf den nächsten Container. Allerdings können Container in Skiplisten auch Zeiger auf andere Container enthalten, welche nicht direkt nachfolgen. Es können also Schlüssel übersprungen werden. Jeder Container hat eine bestimmte Höhe h, welche um 1 kleiner ist als die Anzahl der Zeiger, die ein Container enthält.“

5.2 Natürlicher binärer Suchbaum[9]

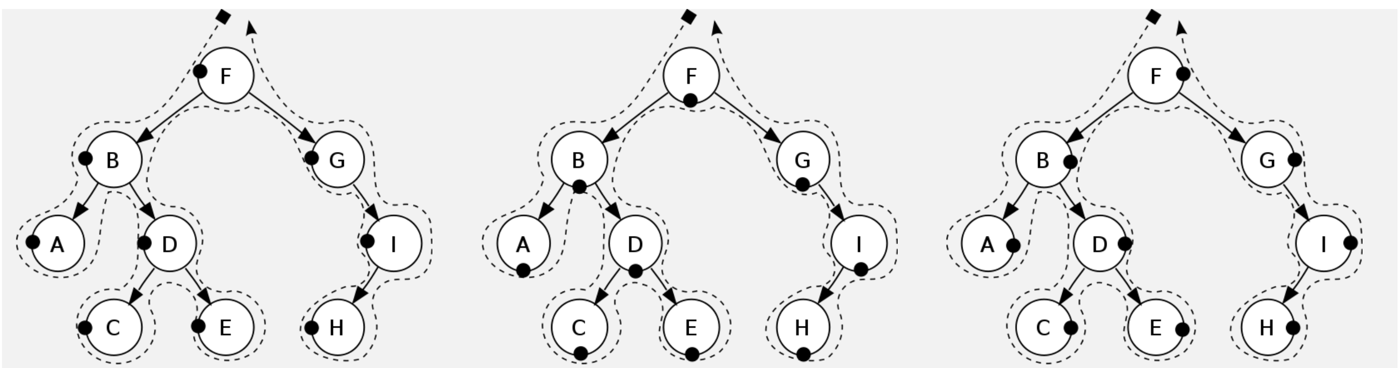


Alles im linken Teilbaum ist kleiner, alles im rechten Teilbaum grösser als die Wurzel. Im Normalfall ist für n innere Knoten und $n+1$ Blätter die Höhe $\mathcal{O}(\log n + 1)$, sie kann aber bei entarteten Bäumen $\mathcal{O}(n)$ sein. Bei einer Suche wird daher, ausgehend von der Wurzel, der Baum mit Hilfe der Knoten als Wegweiser für „links/rechts“ traversiert. Bei einer Einfügeoperation wird zuerst die Einfügepunkt bestimmt (Suche) und dann das Blatt im richtigen Teilbereich eingefügt. Falls beim Löschen eines Knotens noch Kinder dran hängen, müssen diese entsprechend umgeordnet werden (bei einem Kind, wird der Wert ersetzt, bei zwei Kindern rutscht der in-order Nachfolger oder Vorgänger an diesen Platz und nun wird der restliche Teilbaum umgehängt) .

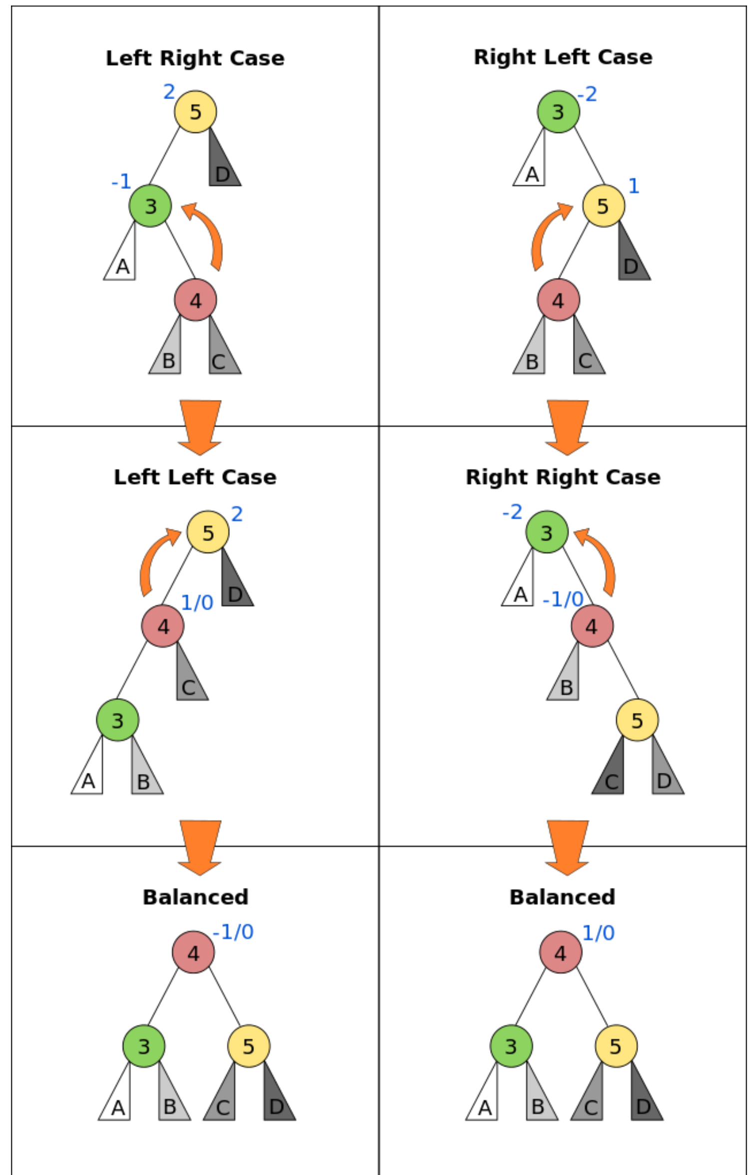
5.3 Traversieren

PRE-ORDER (HAUPTREIHENFOLGE)	IN-ORDER (SYMMETRISCHE RF)	POST-ORDER (NEBENREIHENFOLGE)
1. Visit the root. 2. Traverse the left subtree. 3. Traverse the right subtree	1. Traverse the left subtree. 2. Visit the root. 3. Traverse the right subtree	1. Traverse the left subtree. 2. Traverse the right subtree. 3. Visit the root.
Pre-order: F, B, A, D, C, E, G, I, H	In-order: A, B, C, D, E, F, G, H, I	Post-order: A, C, E, D, B, H, I, G, F

[7] http://summaries.stefanheule.com/en/
[8] Amortisierte Laufzeiten
[9] Bild: http://people.bath.ac.uk/mir20/images/bst_overlap_big.png



5.4 AVL Bäume



"Der AVL-Baum ist ein binärer Suchbaum mit der Eigenschaft, dass sich an jedem Knoten die Höhe der beiden Teilbäume um höchstens eins unterscheidet. Diese Eigenschaft lässt seine Höhe nur logarithmisch mit der Zahl der Schlüssel wachsen und macht ihn zu einem balancierten binären Suchbaum."

$\text{bal}\, p \coloneqq t_{\text{rechts}} - t_{\text{links}} \in \{-1{,}0{,}1\}$ ist der Balancewert, welcher beim Einfügen auch kurzzeitig $\in \{0, \pm 1, \pm 2\}$ sein kann.

Die Einfügelogik ist im Bild nebenan dargestellt (Wikipedia). Da eventuell nicht nur der Knoten, an welchem eingefügt wurde, überprüft werden muss, sondern auch noch weitere, wird die Prozedur *upin* aufgerufen, die alle Knoten bis zur Wurzel auf den Balancewert prüft.

- **XX** Amortisierte Analyse des Einfügens (Kapitel 5.2)

5.5 Selbstanordnung in linearen Listen

- **Move-to-front (MTF)** Mache ein Element zum ersten Element der Liste, nachdem auf das Element (als Ergebnis einer erfolgreichen Suche) zugegriffen wurde. Die relative Anordnung der übrigen Elemente bleibt unverändert.
- **Transpose** Vertausche ein Element mit dem unmittelbar vorangehenden, nachdem auf das Element zugegriffen wurde.
- **Frequency Count** Ordne jedem Element einen Häufigkeitszähler zu, der anfangs 0 ist und die Anzahl der Zugriffe auf das Element speichert. Nach jedem Zugriff auf ein Element wird dessen Häufigkeitszähler um 1 erhöht. Ferner wird die Liste nach jedem Zugriff neu geordnet und zwar so, dass die Häufigkeitszähler der Elemente in absteigender Reihenfolge sind.

5.6 Kompetitive und amortisierte Analyse

„Competitive analysis is a method invented for analyzing online algorithms, in which the performance of an online algorithm (which must satisfy an unpredictable sequence of requests, completing each request without being able to see the future) is compared to the performance of an optimal offline algorithm that can view the sequence of requests

in advance. An algorithm is competitive if its competitive ratio—the ratio between its performance and the offline algorithm's performance—is bounded.”[10]

“Will man nicht die Kosten einer einzelnen Operation abschätzen, sondern etwas über eine Folge von Operationen sagen, so eignet sich eine amortisierte Analyse. Dabei wählt man eine Potentialfunktion ϕ_i welche dem Zustand des zu untersuchenden Algorithmus nach der Operation i einen Wert zuweist.

Die i-te Operation habe tatsächliche Kosten von t_i. Dann sind die amortisierten Kosten dieser Operation gegeben als $a_i = t_i + \phi_i - \phi_{i-1}$. Also gilt für eine Folge von m Operationen:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \phi_i - \phi_{i-1} = \left(\sum_{i=1}^{m} t_i\right) + \phi_m - \phi_0$$

Wenn also ϕ stets positiv ist, und zu Beginn mit 0 initialisiert wird, können damit die tätsächlichen Kosten durch die amortisierten abgeschätzt werden.“[11]

5.7 Optimale Suchbäume (mit DP)

Mittels dem Optimalitätsprinzip der DP kann die Aussage gemacht weerden, dass jeder Teilbaum eines optimalen Suchbaumes selbst ein optimaler Suchbaum ist. Optimal beduetet hier, dass der Suchbaum hinsichtlich der Zugriffshäufigkeiten optimal ist. Dies läuft in $\mathcal{O}(n^3)$.[12]

5.8 Splay Trees

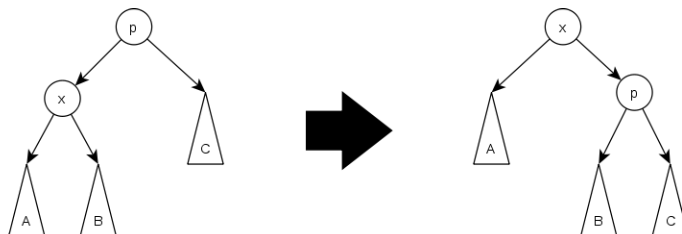
„Analog entspricht der Move-to-front für lineare Listen die folgende Move-to-root-Strategie für binäre Suchbäume: Nach jedem Zugriff auf einen Schlüssel wird er durch Rotationen solange hinauf bewegt, bis er bei der Wurzel angekommen ist.“[13]

Nachfolgende Abschnitte: Wikipedia

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $\mathcal{O}(\log n)$ amortized time.

All normal operations on a binary search tree are combined with one basic operation, called **splaying**. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.

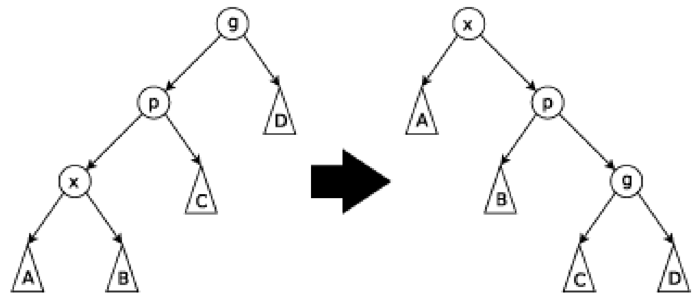
Splay Wird die Splay-Operation auf ein Element *x* in einem Baum *T* angewendet, so sorgt sie dafür, dass *x* nach der Operation in der Wurzel von *T* steht. Dies wird erreicht, indem das Element Schritt für Schritt im Baum hinaufrotiert wird, bis es schließlich bei der Wurzel angekommen ist. Hierzu wird *x* jeweils mit seinem Vater bzw. Großvater verglichen. Aufgrund dieses Vergleiches werden insgesamt sechs Fälle unterschieden, von denen jeweils die Hälfte symmetrisch sind.



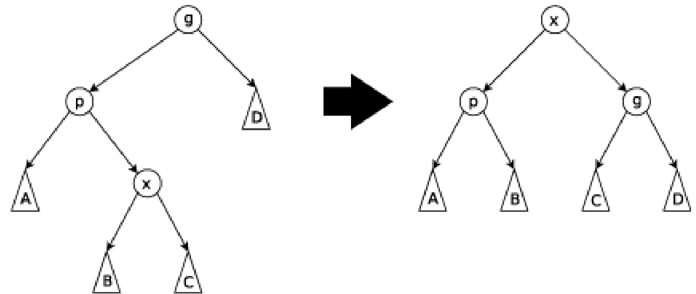
Zick-Rotation Falls *x* das linke Kind seines Vaters ist und keinen Großvater hat, und somit bereits direkt unter der Wurzel steht, wird eine **zick**-Rotation (Rechts-Rotation) durchgeführt. Nun ist *x* die neue Wurzel des Baumes und die Splay-Operation beendet. Liegt *x* im rechten Teilbaum seines Vaters, wird analog eine *zack*-Rotation (Links-Rotation) durchgeführt. Hat *x* einen Großvater, so können zwei Einzelrotationen zu einer Kompositrotation zusammengesetzt werden.

[10] Wikipedia
[11] Stefan Heule
[12] Siehe auch https://forum.vis.ethz.ch/showthread.php?11190-Serie-8-Aufgabe-3-Optimale-Suchb%E4ume&p=163846&viewfull=1#post163846
[13] Buch zur Vorlesung

Zick-Zick-Rotation Ist *x* das linke Kind seines Vaters, welcher das linke Kind des Großvaters von *x* ist, so wird eine **zick-zick**-Rotation (zwei Rechts-Rotationen) durchgeführt. Hierbei wird x mit dem Großvater vertauscht und alle weiteren Unterbäume werden an die entsprechenden Stellen gesetzt. Falls *x* nach der Rotation noch nicht in der Wurzel des Baumes steht, so wird weiterrotiert. Symmetrisch hierzu die *zack-zack*-Rotation, falls *x* das rechte Kind seines Vaters ist, welcher das rechte Kind des Großvaters von *x* ist.



Zack-Zick-Rotation Ist *x* das zweite Kind (von links) seines Großvaters, so wird eine **zack-zick**-Rotation (Links-Rotation gefolgt von einer Rechts-Rotation) durchgeführt. Hierbei tauscht *x* die Position mit seinem Großvater und alle weiteren Unterbäume werden an die entsprechenden Stellen gesetzt. Falls *x* nach der Rotation noch nicht in der Wurzel des Baumes steht, so wird weiterrotiert. Symmetrisch hierzu die *zick-zack*-Rotation, falls *x* das linke Kind seines Vaters ist, welcher das rechte Kind des Großvaters von *x* ist.



Suchen Um ein Element *x* im Baum *T* zu suchen, führt man einfach *splay(x,T)* aus. Dies bewirkt, dass falls *x* in *T* enthalten war, es nun in der Wurzel steht. Somit muss man nur noch die neue Wurzel mit *x* vergleichen. Sind sie unterschiedlich, war *x* nicht im Baum.

Einfügen Um ein Element *x* in einen Splay-Baum *T* einzufügen, sucht man zuerst wie in einem Binärbaum nach *x*. Nachdem diese Suche erfolglos endet, bekommt man den Knoten *v*, an dem *x* angehängt werden müsste. Dieser Knoten *v* wird jetzt mit der *splay*-Operation an die Wurzel gebracht. Somit ist *v* nun an der Wurzel und hat zwei Teilbäume *T1* und*T2*. Jetzt wird die *split*-Operation ausgeführt:

x wird mit *v* verglichen:

- wenn *x* größer als *v*, dann wird *v* mit seinem linken Teilbaum *T1* links an *x* angehängt. Der rechte Teilbaum *T2*wird rechts an *x* angehängt.
- wenn *x* kleiner als *v*, dann wird *v* mit seinem rechten Teilbaum *T2* rechts an *x* angehängt. Der linke Teilbaum *T1*wird links an *x* angehängt.

Somit ist *x* an der Wurzel und an der richtigen Stelle.

Löschen Um *x* aus *T* zu löschen, führt man erst einmal eine Suche auf *x* aus, wird das Element gefunden, wird es gelöscht, und der Unterbaum an den Elternknoten P(x) angehängt. Gefolgt von *splay(P(x), T)*, welches den Elternknoten in die Wurzel holt.

Vereinigen Die Operation *join* vereinigt zwei Splay-Bäume T_1 und T_2, welche unmittelbar vorher mittels *split* getrennt wurden. Hierbei wird zuerst mittels splay(∞,T_1) das maximale Element x_1 des ersten Baumes gesucht und in die Wurzel rotiert. Da die beiden Bäume T_1 und T_2 das Ergebnis einer vorherigen *split*-Operation sind, sind alle Elemente in T_2größer als die Elemente in T_1, weswegen man den Baum T_2 nun ohne Probleme zum rechten Kind von x_1 machen kann.

Aufsplitten Um einen Splay-Baum *T* bei dem Knoten *x* in zwei Splay-Bäume aufzusplitten, macht man zuerst *x* mittels *splay* zur Wurzel von *T*. War *x* im Baum enthalten, kann man nun die Verbindung zu einem der beiden Teilbäume einfach trennen. Steht nach der Splay-Operation ein anderes Element *y* in der Wurzel, so war *x* selbst nicht in *T* enthalten. Ist *x* nun kleiner als *y*, so kann man das linke Kind von *y* abschneiden, andernfalls sein rechtes.

6 Dynamische Programmierung

„Die Induktion bei der Lösung des Rucksackproblems beruht darauf, dass man (eben induktiv) auf Lösungen „kleinerer Instanzen“ desselben Problems zurückgreifen kann. “Kleinere Instanzen” sind problemabhängig definiert, haben aber typischerweise kleinere Parameterwerte. Entscheidend für die Induktion ist, dass wir für eine kleinere Instanz bereits eine optimale Lösung (in der Induktion) berechnet haben und im Induktionsschritt verwenden können. Das geht nur,

wenn sich die optimale Lösung auch tatsächlich aus Teilen ermitteln lässt, die ihrerseits optimale Lösungen des jeweiligen Teilproblems sind. Man nennt dies das Optimalitätsprinzip, das Verfahren des induktiven Ausfüllens einer entsprechenden Tabelle die dynamische Programmierung.“[14]

Eine vollständige Beschreibung eines dynamischen Programms behandelt immer die folgenden Aspekte:[15]

1. Definition der DP-Tabelle: Welche Dimensionen hat die Tabelle? Was ist die Bedeutung jedes Eintrags?
2. Berechnung eines Eintrags: Wie berechnet sich ein Eintrag aus den Werten von anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?
3. Berechnungsreihenfolge: In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden?
4. Auslesen der Lösung: Wie lässt sich die Lösung am Ende aus der Tabelle auslesen?

Die Laufzeit eines dynamischen Programms berechnet sich üblicherweise einfach aus der Grösse der Tabelle multipliziert mit dem Aufwand, jeden Eintrag zu berechnen. Manchmal überwiegt jedoch auch der Aufwand um die Lösung auszulesen.

6.1 Longest common subsequence[16]

„ The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two). (Note that a subsequence is different from a substring, for the terms of the former need not be consecutive terms of the original sequence.)”[17]

$$LCS(n,m) = \begin{cases} LCS(n-1,m-1)+1, & \text{falls } a_n = b_m \\ \max\{LCS(n-1,m), LCS(n,m-1)\}, & \text{sonst} \end{cases}, \qquad \text{mit } LCS(0,m) = LCS(n,0) = 0$$

6.2 Editierdistanz (Levenshtein-Distanz)

„Die Levenshtein-Distanz (auch Editierdistanz) zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln.“[18]

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} + 0, & \text{falls } u_i = v_j \ (\nwarrow) \\ D_{i-1,j-1} + 1, & \text{Ersetzung } (\nwarrow) \\ D_{i,j-1} + 1, & \text{Einfügung } (\leftarrow) \\ D_{i-1,j} + 1, & \text{Löschung } (\uparrow) \end{cases}, \qquad \text{wobei } m = |u|, n = |v|, D_{0,0} = 0, D_{i,0} = \underbrace{i}_{1 \le i \le m}, D_{0,j} = \underbrace{j}_{1 \le j \le n}$$

6.3 Matrixkettenmultiplikation

„Matrix-Kettenmultiplikation bezeichnet die Multiplikation von mehreren Matrizen. Da die Matrizenmultiplikation assoziativ ist, kann man dabei beliebig klammern. Dadurch wächst die Anzahl der möglichen Berechnungswege exponentiell mit der Länge der Matrizenkette an. Mit der Methode der dynamischen Programmierung kann die Klammerung der Matrix-Kette optimiert werden, so dass die Gesamtanzahl arithmetischer Operationen minimiert wird. Der Algorithmus hat eine Laufzeit von $\mathcal{O}(n^3)$.“

6.4 Matrixmultiplikation nach Strassen

Der Algorithmus basiert auf Blockmatrizen und nutzt einige Matrixeigenschaften aus und ist (v.a.) bei grossen Matrizen effizient. Dargestellt ist nur ein Ansatz.

$$C = A \cdot B \Leftrightarrow \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} & A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} & A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \end{pmatrix}, \quad A,B,C \in R^{2^k \times 2^k}$$

6.5 Subset sum

Für eine Menge von Zahlen $A = \{1_1, \dots, a_n\}$ und eine fixe Zahl c ist eine Teilmenge $I \subseteq A$ zu finden, sodass $\sum_{i \in I} a_i = c$ gilt. Falls jede mögliche Teilmenge von A betrachtet wird (indem für jede Zahl im Entscheidungsbaum die beiden Möglichkeiten „nehmen“ und „nicht nehmen“ möglich sind), beträgt die Laufzeit $\mathcal{O}(2^n)$.

---

[14] Buch zur Vorlesung

[15] Blatt 7, Übung zur Vorlesung

[16] https://www.youtube.com/watch?v=P-mMvhfJhu8

[17] Wikipedia

[18] Wikipedia

„Ein anderer Trick ist da schon nützlicher: Man teilt A in zwei gleich grosse Teile $A_1 = \{a_1, \ldots, a_{n/2}\}, A_2 = \{a_{n/2+1}, \ldots, a_n\}$ und berechnet explizit die Menge $T1$ aller erreichbaren Teilsummen für $A1$ sowie die Menge $T2$ aller erreichbaren Teilsummen für $A2$. Dann sortiert man $T1$,sowie $T2$, und muss jetzt lediglich noch jeweils eine Zahl aus $T1$ wie auch aus $T2$ suchen, die sich zu c addieren. Dazu kann man, ähnlich wie beim merge im Mergesort, $T1$ in aufsteigender und simultan $T2$ in absteigender Reihenfolge inspizieren und die Summe s der beiden inspizierten Zahlen mitzvergleichen: Ist s kleiner als c, so inspiziert man die nächstgrössere Zahl in $T1$, und ist s grösser als c,so nimmt man die nächstkleinere Zahl in $T2$. Ist $s = c$, so weiss man, dass c zerreichbar ist. Wenn bis zum Schluss nie $s = c$ gilt, so weiss man, dass c nicht erreichbar ist. Es reicht also, $T1$ und $T2$ einmal linear zu durchlaufen, um die Antwort zu finden. Laufzeit: $\mathcal{O}\left(n\sqrt{2}^n\right)$"[19]

So werden u.U. einige Entscheidungen mehrfach getroffen, daher bietet es sich an, Teillösungen zu berechnen und diese wiederzuverwenden. Wird dies als Matrix dargestellt, so ist die Laufzeit $\mathcal{O}(n \cdot c)$, was, abhängig von c, kleiner oder grösser als $\mathcal{O}(2^n)$ ist (dementsprechend auch nicht, welches Verfahren schneller ist", daher wird die Laufzeit pseudopolynomiell genannt.

6.6 Rucksackproblem[20]

„Aus einer Menge von Objekten, die jeweils ein Gewicht und einen Nutzwert haben, soll eine Teilmenge ausgewählt werden, deren Gesamtgewicht eine vorgegebene Gewichtsschranke nicht überschreitet. Unter dieser Bedingung soll der Nutzwert der ausgewählten Objekte maximiert werden."[21]

Ein erster Ansatz besteht aus einer DP-$n \times g$-Matrix (n Gegenstände, g ist das Maximumgewicht, w_i ist der Wert von Gegenstand i) und läuft in $\mathcal{O}(n \cdot g)$, wobei:
$maxwert(i,g) = \max\{maxwert(i-1,g), w_i + maxwert(i-1, g-g_i)\}$; $maxwert(0,g) = maxwert(i,0) = 0$

Ein zweiter Ansatz dreht das Problem um: Dabei wird nach der mindestens benötigten Gewichtsgrenze, um mit den ersten i Gegensätnden den besten Wert zu erreichen, gesucht, $\mathcal{O}(n \cdot w)$.

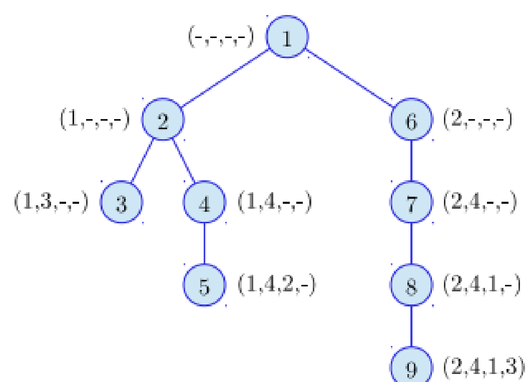
$$\min\left\{mingewicht(i-1,w), mingewicht\left(i-1, \underbrace{w-w_i}_{\text{falls}\geq 0}\right) + g_i\right\}$$

Bei einem dritten Ansatz wird das Resultat nur annähernd ermittelt, sprich ein Gütefaktor ε verwendet in dem alle Werte skaliert und gerundet werden mit k, $w_i' \coloneqq \left\lfloor \frac{w_i}{k} \right\rfloor$ wodurch die Laufzeit $\mathcal{O}\left(\frac{n^3}{\varepsilon}\right)$ erreicht wird. „Das ist ein Polynom sowohl in n als auch in $\frac{1}{\varepsilon}$; man nennt solch ein Verfahren ein voll polynomielles Approximationsschema (**FPTAS**, fully polynomial time approximation scheme)."[22]

7 Backtracking, Branch-and-Bound

7.1 n-Queens

„Es sollen sich keine zwei Damen die gleiche Reihe, Linie oder Diagonale teilen. Im Mittelpunkt steht die Frage nach der Anzahl der möglichen Lösungen."[23]



Ein Ansatz ist, die Lösung als Vektor der Länge n darzustellen, wobei die i-te Komponente die Zeilennummer der Dame der i-ten Spalte enthält. „[Beim Durchiterieren] versuchen wir, eine aktuelle Teillösung so zu erweitern, dass die neu entstandene Teillösung noch immer in sich konsistent ist. Kann eine Lösung nicht erweitert werden, dann gehen wir zurück und probieren die alternativen Erweiterungen (sofern existent, ansonsten gehen wir noch weiter zurück)."[24] Dies nennt sich **Backtracking**.

[19] Buch zur Vorlesung
[20] Verwandt mit Subset sum
[21] Wikipedia
[22] Buch zur Vorlesung
[23] Wikipedia
[24] Vorlesungs-Moodle, auch Quelle für Grafik

Das gleiche Prinzip lässt sich auch beim Erfüllbarkeitsproblem der Aussagenlogik (**SAT**), bei dem bestimmt wird, ob eine aussagenlogische Formel (in CNF-Form) erfüllbar ist, anwenden.

7.2 Branch-and-bound[25]

Wir haben im vorigen Abschnitt eine Technik kennengelernt, die es erlaubt, aussichtslose Teillösungen frühzeitig zu verwerfen. Allerdings haben wir sie nur für Entscheidungsprobleme angewendet, bei denen am Ende die Antwort Ja oder Nein steht. Für Optimierungsprobleme ist Branch-and-Bound eine ähnliche Vorgehensweise, allerdings müssen wir überlegen, wann eine Teillösung nicht weiter verfolgt werden soll. O.B.d.A. betrachten wir Maximierungsprobleme, sofern nicht anders genannt. Wie vorher speichert jeder Knoten des Ablaufbaums eine Teillösung. Zusätzlich wird in jedem Knoten eine obere Schranke gespeichert. Diese gibt an, welchen Wert eine gültige (ggf. mehrfache) Erweiterung der an diesem Knoten repräsentierten Teillösung maximal erreichen kann. Für den Ablaufbaum verwalten wir ausserdem noch eine untere Schranke. Diese ist das Maximum der bereits gefundenen Lösungen und einer globalen unteren Schranke (sofern existent). Stellen wir an einem Knoten fest, dass die obere Schranke kleiner gleich der unteren Schranke ist, dann muss die an diesem Knoten repräsentierte Teillösung nicht weiter verfolgt werden (denn keine Erweiterung kann einen besseren Wert als die derzeit beste Lösung erreichen). Um Branch-and-Bound anzuwenden, müssen zusätzlich drei Module spezifiziert werden:

1. Branch: Entscheide, wie eine gegebene Teillösung zu erweitern ist.
2. Search: Welche aktuelle Teillösung soll als nächstes betrachtet und ggf. erweitert werden?
3. Learn: Was folgt aus den bisherigen Erkenntnissen für erzwungene Teile der Lösung?

Im Allgemeinen wählt das Search-Modul im Ablaufbaum ein Blatt v mit der höchsten oberen Schranke aus. Ist der Wert der dort gespeicherten Teillösung s_v grösser als die bisherige untere Schranke, so wird diese entsprechend aktualisiert. Ist s_v eine vollständige Lösung, deren Wert mit der unteren Schranke zusammenfällt, dann wird s_v als aktuelles Optimum abgespeichert. Das Branch-Modul versucht nun, den Knoten v zu erweitern. Das gesamte Verfahren terminiert, sobald die obere Schranke jedes Blatts maximal so gross wie die untere Schranke ist. Zur Erinnerung: Die untere Schranke gibt die beste bisher erreichte Lösung an. Haben nun alle Blätter eine obere Schranke, die maximal so gross wie die untere Schranke ist, dann kann keine Teillösung so erweitert werden, dass eine bessere als die bisher gefundene Lösung entsteht. Die Benutzung des Learn-Moduls ist nicht zwingend, kann aber zu einer besseren Laufzeit führen, da u.U. wesentlich weniger (Teil)lösungen betrachtet werden müssen.

Dieser Ansatz kann für eine Maximierung des SAT-Problems (vorheriger Abschnitt) oder für das Rucksackproblem[26] verwendet werden. Ein Ansatz dabei ist ein Greedy-Algorithmus, der zuerst die Objekte absteigend nach dem Wert sortiert und dann solange Objekte einpackt, wie möglich.

- **(XX)** Knapsack

8 Graphenalgorithmen

$$G = \big(\underbrace{V}_{\text{Knoten}}, \underbrace{E}_{\text{Kanten}}\big), \quad \underbrace{E \subseteq V \times V}_{\text{gerichtet}} \text{ oder } \underbrace{E \subseteq \{\{v,w\} \in V\}}_{\text{ungerichtet}}, \quad n \coloneqq |V|, m \coloneqq |E|$$

Ein Graph kann als Adjazenzmatrix ($A_{ij} = 1 \Leftrightarrow (i,j) \in E$) oder als Adjanzenzliste gepsichert werden. Beide Möglichkeiten haben andere Vor- und Nachteile.

8.1 Reflexive und transitive Hülle

Eine transitive Hülle bedeutet, dass sofern $a \to b$ und $b \to c$ auch $a \to c$ gilt. Reflexiv heisst, es gibt eine Schleife von a nach a.

8.2 Topologisches Sortieren

„Topologische Sortierung bezeichnet in der Mathematik eine Reihenfolge von Dingen, bei der vorgegebene Abhängigkeiten erfüllt sind.

[25] Beschreibung und Erklärung aus Vorlesungs-Moodle

[26] Siehe auch http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec16.pdf

Anstehende Tätigkeiten einer Person etwa unterliegen einer Halbordnung: es existieren Bedingungen wie „Tätigkeit A muss vor Tätigkeit B erledigt werden". Eine Reihenfolge, welche alle Bedingungen erfüllt, nennt man topologische Sortierung der Menge anstehender Tätigkeiten. Im Gegensatz zur Sortierung einer Totalordnung ist die Reihenfolge nicht eindeutig, sondern es kann mehrere Möglichkeiten geben. Wenn gegenseitige Abhängigkeiten bestehen, ist eine topologische Sortierung unmöglich."[27]

„Wenn man stets einen Knoten mit in-degree 0 (also einen, ohne eingehende Kante) aus dem Graphen entfernt, und diesem die nächstgrössere Nummer gibt, erhält man schlussendlich eine topologische Sortierung. Eine konkrete Implementierung könnte folgendermassen aussehen: Jeder Knoten erhält einen Zähler, welcher die Anzahl eingehender Kanten quantifiziert. Um diesen Zähler zu initialisieren, benötigt man $\mathcal{O}(n+m)$ Zeit. Zusätzlich verwaltet man eine lineare Liste mit allen Knoten, deren Zähler 0 ist. Nun kann man, solange diese Liste noch Elemente enthält, ein beliebiges entfernen, und diesem die nächstgrössere Nummer zuweisen. Zusätzlich folgt man allen Pfeilen dieses Knotens (was mit der Adjazenzlistenrepräsentation sehr einfach ist) und verkleinert die Zähler aller dieser Knoten. Wird ein Zähler 0, so wird dieser in die Liste aufgenommen. Da jeder Knoten und jede Kante höchstens einmal bearbeitet wird, ergibt sich so eine totale Laufzeit von linearer Grösse im Input: $\mathcal{O}(n+m)$."[28]

8.3 Traversieren

8.3.1 DFS, $O(|E|) = O(m)$

"One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking."[29]

DFS kann für die topologische Sortierung und den Test, ob ein Graph zusammenhängend ist, verwendet werden.

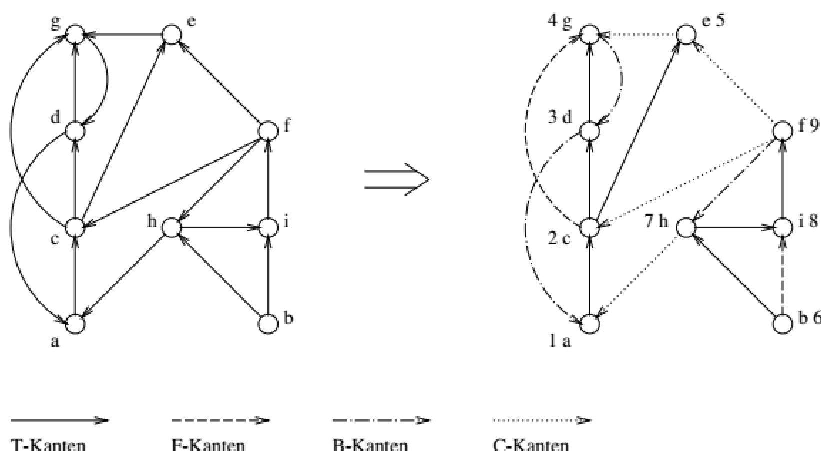


Abbildung 2.7.1: Ein Beispiel für den DFS-Algorithmus auf gerichteten Graphen

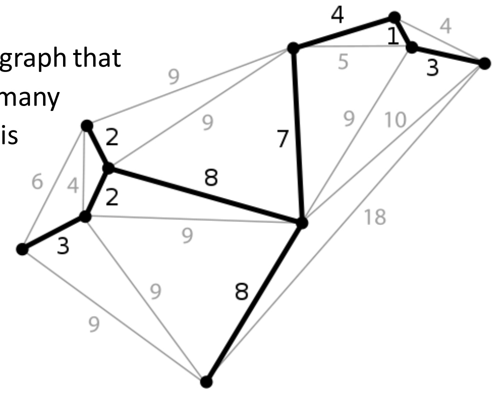
- **T-Kante (Tree)** Endknoten v wurde noch nie besucht. T-Kanten spannen immer Bäume auf.
- **F-Kante (Forward)** Endknoten v wurde schon mal besucht und liegt in der Besuchsreihenfolge später als u. Es gibt immer einen reinen T-Weg von u nach v.
- **B-Kante (Backward)** Endknoten v wurde schon mal besucht und liegt in der Besuchsreihenfolge vor u, die Tiefensuche in v ist aber noch nicht abgeschlossen. B-Kanten erzeugen Kreise im Graphen, es gibt immer einen reinen T-Weg von v zurück nach u.
- **C-Kante (Cross)** Keinen der obigen Fälle. C-Kanten verbinden unabhängige Teilbäume.[30]

8.3.2 BFS, $O(|E|) = O(m)$

"BFS is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on."

[27] Wikipedia
[28] Stefan Heule
[29] Wikipedia
[30] Grafik und Erklärungen via Samuel Steffen, TA

8.4 Minimaler spannender Baum (MST)



“Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree.”[31]

8.4.1 Greedy[32]

Jede Kante hat zu jedem Zeitpunkt einen Status. Entweder ist sie gewählt, verworfen oder noch unentschieden, wobei zu Beginn alle Kanten unentschieden sind. Weiter soll während des ganzen Verfahrens folgende Invariante gelten:

Falls es einen MST T von G gibt, enthält dieser alle gewählten, aber keine verworfenen Kanten.

Ein Schnitt in einem Graphen $G = (V, E)$ sei nun eine Aufteilung von V in S und $\bar{S} = V \setminus S$. Eine Kante $e \in E$ kreuzt diesen Schnitt nun, wenn der eine Endpunkt in S und der andere in $\bar{S}$ liegt. Das Verfahren befolgt nun ständig eine der folgenden Regeln, bis der MST gefunden wurde:

1. Betrachte einen Schnitt, den keine schon gewählte Kante kreuzt. **Wähle** eine kürzeste Kante unter allen unentschiedenen Kanten, welche diesen Schnitt kreuzen.
2. Wähle einen einfachen Zyklus (einen Zyklus also, der sich selbst nicht „kreuzt“), der keine verworfene Kante enthält. **Verwerfe** unter den unentschiedenen Kanten die längste.

8.4.2 Kruskal (mit Union-Find) [33]

Es werden alle Kanten in aufsteigend sortierter Reihenfolge betrachtet. Verbindet die gerade betrachtete Kante zwei unabhängige Teilbäume, so nehmen wir die Kante. Andernfalls, also wenn beide Endpunkte im selben Teilbaum liegen, verwerfen wir die Kante. Hinweis: Zu Beginn gibt es n Teilbäume der Grösse 1, bestehend aus nur einem Knoten. Benötigt wird also eine einfach Union-Find-Datenstruktur, um den Algorithmus von Kruskal zu implementieren. $\mathcal{O}(n \cdot \log n)$

8.4.3 Prim/Dijkstra[34]

Eine weitere Möglichkeit, dieses Verfahren zu implementieren benutzt nur die Regel 1. Es werden also nirgends Kanten explizit ausgeschlossen: Man verwaltet eine Menge von erledigten Knoten, wobei es nur gewählte Kanten zwischen erledigten Knoten gibt. Damit gibt es immer einen Schnitt zwischen den erledigten und unerledigten Knoten, wobei keine gewählte Kante diesen Schnitt überquert. Folglich kann also die kürzeste Kante, welche diesen Schnitt kreuzt, gewählt werden.

Konkret wird nicht die Menge der erledigten Knoten verwaltet, sondern man hält alle unerledigten Knoten, die aus V' erreichbar sind, in einem Fibbonacci Heap. Dann kann man jeweils das Minimum aus dem Heap entfernen und diese Kante zum MST hinzufügen. Zudem muss man nun alle Nachbarn b des gerade betrachteten Knotens a ansehen, wobei es zwei Fälle gibt:

1. Der Knoten b ist bereits im Heap, aber der Wert von b im Heap ist grösser, als das Gewicht der Kante (a, b). Dann muss der Wert von b im Heap verringert werden.
2. Der Knoten b ist nicht im Heap, und ist nun (neu) erreichbar aus V'. b wird also eingefügt, mit den Wert der Kante (a, b).

8.5 Union-Find[35]

Es soll eine Menge von Mengen verwaltet werden, wobei jede Menge einen „Namen“ trägt. Oftmals wir ein kanonisches Element ausgewählt, welches den Namen der Menge bestimmt. Weiter sollen drei Operationen möglich sein:

---

[31] Wikipedia
[32] Stefan Heule
[33] Stefan Heule
[34] Stefan Heule
[35] Stefan Heule

1. $\mathrm{makeSet}(a)$: Erzeugt eine Menge mit dem Element a.
2. $\mathrm{find}(a)$: Findet das Element a und gibt den Namen der Menge, in welcher a vorkommt, zurück.
3. $\mathrm{union}(M, N)$: Vereinigt die Mengen M und N.

Eine mögliche Implementierung sieht folgendermassen aus: Die Mengen werden als allgemeine Bäume gespeichert, wobei jeder Knoten einen Elternzeiger besitzt. Für die Wurzel, welches zugleich das kanonische Element darstellt, zeigt dieser auf das Element selbst. Die drei Operationen sind nun trivial zu implementieren:

1. Erzeuge ein Element mit einem Zeiger auf sich selbst.
2. Gehe zum Element a und folge den Elternzeigern, bis das kanonische Element gefunden wurde.
3. Hänge die eine Wurzel unter die Wurzel der anderen Menge.

Eine Menge zu erzeugen und zwei Mengen zu vereinigen geht in konstanter Zeit. Ein Element zu finden, geht in $\mathcal{O}(h)$ wenn h die Höhe des Baumes bezeichnet. Will man h logarithmisch in n, der gesamten Anzahl Elemente, limitieren, so muss beim Vereinigen nicht wahllos vorgegangen werden. Entweder, man vereinigt nach Höhe, oder nach Grösse:

- **Höhe**: Der kleinere Baum (kleiner in der Höhe) wird unter den grösseren gehängt.
- **Grösse**: Der kleinere Baum (weniger Elemente) wird unter den grösseren gehängt.

Beide Verfahren liefern logarithmische Laufzeit, wobei bei beiden gewisse Informationen bei den Knoten gespeichert werden müssen (Höhe / Grösse des Teilbaumes).

8.6 Fibonacci Heap[36]

Fibonacci Heaps sind für viele Anwendungen, insbesondere für Graphenalgorithmen von theoretischem Interesse. In der Praxis werden sie jedoch nur selten eingesetzt, wegen ihrer Komplexität und weil sie sich hinter grossen Konstanten (Big Oh Notation) verstecken. Der Hauptsächliche Vorteil ist die decrease_key Funktion, welche amortisiert in $\mathcal{O}(1)$ läuft.

8.6.1 Struktur

Ein Fibonacci Heap besteht aus einer Menge von min-heap geordneter Bäume, die in einer ungeordneten, doppelt verketteten Wurzelliste zusammengefasst sind. Jeder Knoten hat vier Zeiger: Es gibt einen Zeiger auf den Elternknoten, und einen auf ein beliebiges seiner Kinder. Zusätzlich sind auch alle Kinder eines Knotens in einer doppelt verketteten Liste verbunden, weshalb jeder Knoten zusätzlich noch zwei Zeiger zu seinen Nachbarn hat. Hat ein Knoten nur ein Kind x so gilt für dieses x.left = x.right = x. Der Elternzeiger von Knoten in der Wurzelliste wird auf NULL gesetzt. Doppelt verkettete Listen geben uns zwei Vorteile: Es ist möglich, Knote aus der Liste zu löschen oder zwei Listen zusammenzuhängen, und zwar in konstanter Zeit.

Weiter gibt es bei jedem Knoten noch zwei Felder: Zum einen wird der Grad, also die Anzahl Kinder in der Kindliste gespeichert. Zudem gibt es ein Feld „mark“, welches angibt, ob ein Knoten ein Kind verloren hat seit dem letzten Mal, als es Kind eines anderen Knotens wurde. Neue Knoten erhalten für dieses Feld *false*.

Der Fibonacci Heap wird über einen Zeiger *min* angesprochen, der auf das minimale Element in der Wurzelliste zeigt, und damit auf das globale Minimum. Ist der Heap leer, ist *min* einfach NULL.

8.6.2 Operationen auf Fibonacci Heaps

8.6.2.1 Heap erstellen, Minimum finden, $\mathcal{O}(1)$ real und amortisiert

Diese beiden Operationen sind trivial und werden nicht genauer erklärt

8.6.2.2 Ein Element einfügen, $\mathcal{O}(1)$ real und amortisiert

Es wird ein neuer Knoten alloziert, sein Grad auf 0 gesetzt, der Eltern- und Kindzeiger seien NULL, er wird nicht markiert und seinen beiden Nachbarzeiger zeigen auf ihn selbst. Nun kann der Knoten in die Wurzelliste eingefügt werden, und gegeben falls der *min*-Zeiger auf das neue Element gesetzt werden.

---

[36] Stefan Heule

8.6.2.3 Union, $\mathcal{O}(1)$ real und amortisiert

Zwei Fibonacci Heaps können einfach vereinigt werden, indem die Wurzellisten der beiden Heaps zusammengehängt werden und das Minimum angepasst wird.

8.6.2.4 extractMin, $\mathcal{O}(D(n)) = \mathcal{O}(log\, n)$ amortisiert

Die Funktion *extractMin* oder *deleteMin* funktioniert in drei Phasen

1. Das minimale Element wird gespeichert (um es später zurückzugeben) und aus der Wurzelliste entfernt. Alle Kinder dieses Knotens werden in die Wurzelliste aufgenommen und deren Elternzeiger wird angepasst.
2. Nun müssen wir den *min*-Zeiger anpassen. Da in der Wurzelliste aber bist zu $\mathcal{O}(\mathrm{n})$ Elemente zu finden sind, werden wir die Wurzelliste zuerst restrukturieren. Wir fordern, dass keine zwei Knoten in der Wurzelliste denselben Grad haben. Dazu benutzen wir ein Array *A* der Länge $D(n)$, wenn $D(n)$ der maximale Grad eines Knotens im Fibonacci Heaps mit *n* Elementen ist. Die Wurzelliste wird nun traversiert, und bei jedem Knoten x wird dieser in A[Grad(x)] gespeichert. Ist dort bereits ein Knoten, verletzt dieser unsere Forderung, und wir verbinden die beiden Knoten zu einem min-Heap. Dazu wird einfach der Knoten mit dem grösseren Schlüssel als Kind unter den anderen Knoten gehängt.
3. Nun muss nur noch aufgeräumt und das neue Minimum bestimmt werden. Dazu wird die Wurzelliste geleert ($min \coloneqq NULL$) und mit dem Array A wieder neu aufgebaut. In selben Durchgang kann auch gleich das neue Minimum gesetzt werden.

8.6.2.5 decreaseKey, $\mathcal{O}(1)$ amortisiert

Um einen Schlüssel zu verkleinern, gehen wir zu diesem Knoten und verändern den Schlüssel. Wird dadurch die Heapbedingung verletzt und wird der Schlüssel kleiner als der Schlüssel des Elternknoten, so trennen wir den Knoten und hängen ihn in die Wurzelliste, wobei wir „mark“ auf *false* setzen. Der Elternknoten wird nun markiert, ausser diese befindet sich in der Wurzelliste. War er bereits markiert, so trennen wir diesen Knoten ebenfalls; so gehen wir rekursiv bis möglicherweise ganz nach oben. Wenn nötig wird nun noch der *min*-Zeiger aktualisiert, wobei diese entweder unverändert bleibt, oder der Knoten wird, dessen Schlüssel wir gerade verändert haben.

8.6.2.6 delete, $\mathcal{O}(D(n)) = \mathcal{O}(log\, n)$ amortisiert

Einen Schlüssel zu Löschen ist einfach: Wir verändern seinen Schlüssel einfach auf minus Unendlich und entfernen dann das Minimum.

8.6.3 Analyse

Für die amortisierte Analyse wird eine Potenzialfunktion benötigt. Hierfür sei $t(H)$ gleich die Anzahl Elemente in der Wurzelliste und mit $m(H)$ werde die Anzahl markierter Elemente im gesamten Heap angegeben. Dann ist die Potenzialfunktion definiert als:

$$\varphi(H) = t(H) + 2m(H)$$

Zu Beginn haben wir einen leeren Heap, das Potenzial ist also 0. Danach ist gemäss Definition nur ein nicht-negatives Potenzial möglich, was uns mit den amortisierten Kosten eine obere Schranke für die Tatsächlichen Kosten gibt.

Es bleibt noch zu zeigen, dass $D(n)$ auch wirklich in $\mathcal{O}(\log n)$ liegt. Dazu schaut man in der Literatur nach.

8.7 Shortest Path Problem

„[It] is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.“[37]

8.7.1 Bellman-Ford[38]

Mit einer Laufzeit $\mathcal{O}(m \cdot n) = \mathcal{O}(n^3)$ ist der Algorithmus von Ford-Bellmann zwar nicht der Schnellste, insbesondere langsamer als Dijkstra’s Lösung, dafür aber können Kantengewichte auch negativ sein. Gibt es negative Zyklen, so ist

[37] Wikipedia
[38] Stefan Heule

der Begriff „kürzester Pfad“ zwar nicht sinnvoll zu definieren, mit dem Algorithmus von Ford-Bellmann lassen sich solche negativen Zyklen jedoch erkennen. Wie bei Dijkstra werden auch hier alle kürzesten Pfade von einem Startknoten aus gefunden.

Das Verfahren ist sehr simpel: Es gibt ein Array d , indem die kürzeste Distanz zum jeweiligen Knoten gespeichert wird. Nun wird einfach $n-1$ mal über alle Kanten iteriert, wobei mit zuerst nur Pfade der Länge 1, dann 2 usw. betrachtet werden. Da es höchstens $n-1$ Kanten in einem Pfad haben kann, liefert der Algorithmus stets das korrekte Ergebnis. Will man zusätzlich wissen, ob es negative Zyklen gibt, iteriert man bis n. Verändert sich in diesem letzten Schritt noch etwas, gibt es mindestens einen negativen Zyklus.

Es ist einfach, auch gleich noch den kürzesten Pfad zu berechnen, anstelle nur die Länge dieses Pfades. Dazu speichert man einfach in einem zweiten Array p den Vorgänger jedes Knotens.

8.7.2 Dijkstra[39]

Der Algorithmus von Dijkstra dient der Berechnung aller kürzesten Pfade von einem einzelnen Startknoten aus in einem kantengewichteten Graphen. Die funktioniert aber nur bei positiven Kantengewichten.

Es wird dabei folgendermassen vorgegangen: Man verwaltet einen „Rand“ und die Menge der bereits besuchten Knoten. Im Rand befinden sich Knoten, die noch nicht besucht sind, aber von einem besuchten Knoten aus erreicht werden können. Solange der Rand noch nicht leer ist, wird folgendermassen vorgegangen:

- Entferne das Minimum aus dem Rand, denn zu diesem Knoten kommen wir sicher nicht mehr günstiger. Der Knoten wird als besucht markiert.
- Dann werden alle Nachbarn des Knotens inspiziert. Ist der Nachbar bereits im Rand und der Wert des jetzigen Knotens plus das Kantengewicht der Verbindung zusammen kleiner als der Wert des Nachbars, wird der Wert verkleinert. Ist der Wert bereits kleiner oder gleich gross, muss nichts gemacht werden. Ist der Nachbar hingegen noch nicht im Rand, so wird dieser Knoten neu eingefügt, und erhält den Wert des jetzigen Knotens + das Kantengewicht.

Zu Beginn wird der Rand mit den Nachbarn des Startknotens initialisiert. Alternative kann anstelle eines Randes auch die Menge aller unbesuchten Knoten verwaltet werden. Zu Beginn werden dann einfach alle Werte auf unendlich gesetzt.

Die Distanz und ob ein Knoten besucht ist oder nicht kann direkt bei den Knoten gespeichert werden, oder man verwendet ein zusätzliches Array dafür. Um den Rand zu repräsentieren eignet sich am besten ein Fibbonacci-Heap, was in einer schlussendlichen Performance von $\mathcal{O}(m + n \log n)$ resultiert.

8.8 Zunehmende-Wege-Algorithmus nach Ford-Fulkerson

„The idea behind the algorithm is as follows: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Constraints: The flow along an edge can not exceed its capacity. The net flow $u \to v$ must be the opposite of the net flow $v \to u$. The net flow to a node is zero, except for the source, which “produces” flow, and the sink, which “consumes” flow. And flow cannot get lost along an edge.”[40]

Rückwärtspfeile sind möglich. Der Restflussgraph ist der ursprüngliche Graph für welcher die Kapazität die ursprüngliche Kapazität abzüglich des Flusses beträgt. Ferner beinhaltet dieser Graph keinen Fluss.

8.9 Max-Flow-Min-Cut[41]

Ein maximaler Fluss im Netzwerk hat genau den Wert eines minimalen Schnitts.

Sei $G(V, E)$ ein endlicher gerichteter Graph mit den Knoten V und den Kanten E. Jede Kante (u, v) vom Knoten u zum Knoten v habe eine nichtnegative Kapazität $c(u, v)$. Außerdem gibt es einen Quellknoten s, in dem der Netzwerkfluss beginnt, und einen Zielknoten t, in dem der Netzwerkfluss endet.

---

[39] Stefan Heule

[40] Wikipedia

[41] Wikipedia

Ein Schnitt ist eine Aufteilung der Knoten senkrecht zum Netzwerkfluss in zwei disjunkte Teilmengen S und T für die gilt, $s \in S$ und $t \in T$. Die Kapazität eines Schnittes (S, T) ist die Summe aller Kantenkapazitäten von S nach T, also $c(S,T) = \sum_{u \in S, v \in T | (u,v) \in E} c(u, v)$.

Die folgenden drei Aussagen sind äquivalent:

- f ist der maximale Fluss in G.
- Das Residualnetzwerk G_f enthält keinen augmentierenden Pfad.
- Für mindestens einen Schnitt (S, T) ist der Wert des Flusses gleich der Kapazität des Schnittes: $|f| = c(S, T)$

8.10 Max Flow

8.10.1 $\mathcal{O}(nm^2)$ nach Edmonds und Karp

Edmonds-Karb ist identisch mit Ford-Fulkerson, ausser dass der augmenting path anders gesucht wird: es wird der kürzeste Pfad (mit BFS) gesucht. Dies kann erreicht werden, indem die Kapazitäten normiert werden.

8.10.2 $\mathcal{O}(m^2 n)$ nach Dinic

Identisch mit Edmonds-Karp, ausser, dass die Rückwärtskanten so spät wie möglich gewählt werden.

8.11 Matching in bipartiten Graphen. Satz von Hall

$G = (V_1, V_2, E), E \subseteq V_1 \times V_2$, i.e. a graph is **bipartite** if V can be split into two disjoint sets $V_1, V_2; V = V_1 \cup V_2$, such that no edge connects two vertices in the same subset $V_i (i = 1,2)$.[42]

„Es seien X eine endliche Menge und $A_1, \dots, A_n$ Teilmengen von X. Dann sind folgende Aussagen äquivalent:

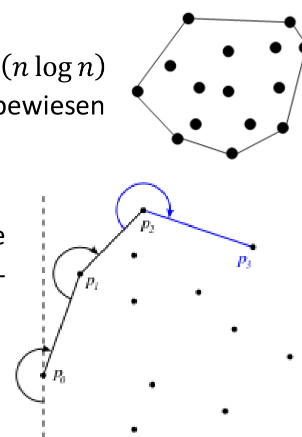
- Es gibt $x_i \in A_i$ derart, dass die $x_1, \dots, x_n$ paarweise verschieden sind.
- Für jede Teilmenge $I \subset \{1, \dots, n\}$ ist $|\bigcup_{i \in I} A_i| \geq |I|$"[43]

9 Geometrische Algorithmen

Jede Beschreibung eines Scanline-Algorithmus sollte die folgenden Aspekte umfassen:[44]

1. **Haltepunkte** In welche Richtung verläuft die Scanline? Was sind die Haltepunkte?
2. **Scanline-Datenstruktur** Welche Objekte muss die Datenstruktur verwalten? Welche Operationen müssen unterstützt werden? Was ist eine angemessene Datenstruktur?
3. **Aktualisierung** Was passiert, wenn die Scanline auf einen neuen Haltepunkt trifft?
4. **Auslesen der Lösung** Wie lässt sich die Lösung auslesen?

9.1 Konvexe Hülle



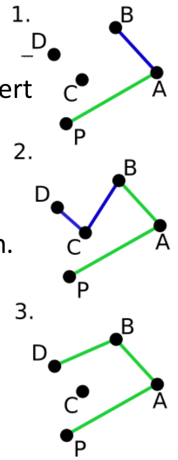
Mittels dem Argument, dass das vergleichsbasierte Sortieren eine untere Schranke von $\Omega(n \log n)$ liegt, kann auch bei Algorithmen für eine konvexe Hülle die untere Schranke von $\Omega(n \log n)$ bewiesen werden.

9.1.1 Jarvis (gift wrapping algorithm)[45]

Man beginnt bei einem Punkt p_0, von welchem man weiss, dass dieser in der konvexen Hülle liegen wird (z.B. der Punkt mit der kleinsten x-Koordinaten). Dann wird iterativ vorgegangen, und ein Punkt p_{i+1} ausgewählt, sodass alle Punkte links von der Geraden p_i, p_{i+1} liegen. Das geht in linearer Zeit, indem man die Winkel aller Geraden von p_0 durch einen andere Punkt betrachtet. Besteht die konvexe Hülle also aus h Punkten, so ergibt dies eine Laufzeit von $\mathcal{O}(nh)$, was in der Praxis ziemlich schlecht ist.

[42] Diskrete Mathematik, U. Maurer
[43] Wikipedia
[44] Blatt 13, Übung zur Vorlesung
[45] Stefan Heule

9.1.2 Graham[46]



Wiederum wird mit einem Punkt p_0, welcher sicher in der konvexen Hülle enthalten ist, begonnen. Nun berechnet man den Winkel zwischen der Gerade p_0, p_i und der Gerade $x = 0$ für alle i aus und sortiert diese aufsteigend. Dies kostet $\mathcal{O}(n \log n)$ Zeit. Dann geht man durch die sortierte Folge der Punkte, beginnend bei p_1. Dabei wird für jeden Punkt entschieden, ob man einen Links- oder Rechtsknick durchgeführt hat. Bei einem Linksknick ist alles ok, wurde jedoch ein Rechtsknick ausgeführt beim Weg von p_{i-2} über p_{i-1} zu p_i, so wurde p_{i-1} fälschlicherweise zur konvexen Hülle hinzugenommen. Folglich wird dieser Punkt gestrichen und man betrachtet p_i erneut (jetzt ist es wieder möglich, dass ein Linksknick gemacht wurde).

Mit diesem Verfahren kann natürlich auch die konvexe Hülle eines Polygons gefunden werden, sogar in linearer Zeit, da dort kein Sortieren notwendig ist, und man direkt dem Polygon folgen kann.

9.1.3 Linearer Scan

- Sortiere jede Hälfte nach x
- Obere Hälfte: immer Rechtskurve
- Untere Hälfte: immer Linkskurve

$$\mathcal{O}(n \log n)$$

9.2 Schnitt orthogonaler Liniensegmente prüfen[47]

Um n orthogonale Liniensegmente auf Schnitte zu überprüfen, kann man das Scanline-Prinzip verwenden. Dazu werden alle Segmente aufsteigend bezüglich ihrer x-Koordinate sortiert, um die Haltepunkte der Scanline zu bestimmen. Um den aktuellen Status der Scanline zu speichern, wird ein binärer Suchbaum (idealerweise ein Blattsuchbaum). Dann wird links begonnen, und die Sweepline bewegt sich fortlaufend nach rechts, wobei an jedem Haltepunkt einer von drei Fällen eintrifft:

- Der Haltepunkt ist linkes Ende eines horizontalen Segmentes: Das Segment wird in die Scanline Datenstruktur aufgenommen, und zwar wird die y-Koordinate des Segmentes in den Baum eingetragen.
- Der Haltepunkt ist rechtes Ende eines horizontales Segmentes: Das Segment wird aus der Scanline-Datenstruktur entfernt.
- Der Haltepunkt ist ein vertikales Segment: Dieses Segment wird nicht in die Datenstruktur aufgenommen, dafür werden aber alle aktiven Segmente (also alle, die sich in der ScanlineDatenstruktur befinden) auf einen Schnitt mit diesem Segment überprüft. Dazu kann ein rangequery mit der oberen und unteren Y-Koordinate als Grenzen verwendet werden.

Will man nun nur wissen, ob ein Schnitt vorhanden ist, so geht das mit einem beliebigen balancierten Suchbaum, denn sobald ein Schnitt gefunden wurde, kann abgebrochen werden.

Sollen jedoch alle Schnitte berichtet werden, benötigt man einen Blattsuchbaum, mit verketteter Blattliste. Dann kann man bei der range-query nach dem einen Ende suchen, und dann einfach so lange der verketteten Liste folgen, bis das andere Ende der query überschritten wurde.

Ist man nur an der Anzahl der Schnitte interessiert, so ist eine weitere Modifikation der ScanlineDatenstruktur nötig. An jedem Knoten werden nun zusätzlich die Anzahl Elemente in diesem Teilbaum gespeichert. Das lässt nun zu, dass für eine range-query in logarithmischer Zeit die Anzahl antworten ermittelt werden können. Dazu sucht man nach beiden Enden, wobei sich diese beiden Pfade an einem eindeutigen Knoten v teilen. Von dort aus folgt man nun zuerst dem linken Pfad, und wenn immer dieser Pfad nach rechts verzweigt, hat der Knoten an der Verzweigung einen linken Sohn, dessen Kinder alle im Bereich liegen. Die Anzahl Elemente, die dort steht, wird also mitgezählt. Dann wird auf dem rechten Pfad symmetrisch dasselbe ausgeführt. Für die Endpunkte selbst ist dann eine Fallunterscheidung nötig, insgesamt geht alles aber in logarithmischer Zeit.

Zusammenfassend gelten also folgende Laufzeiten:

- detect: $\mathcal{O}(n \log n)$

---

[46] Stefan Heule
[47] Stefan Heule

- report: $\mathcal{O}(n \log n + k)$ bei k Schnitten, mit $k = \mathcal{O}(n^2)$
- count: $\mathcal{O}(n \log n)$

Für orthogonale Segmente waren also primär eine Datenstruktur nötig, nämlich ein binärer Suchbaum (balanciert, möglicherweise Blattsuchbaum). Dazu wurde noch die sortierte Liste der Haltepunkte benötigt, diese konnte jedoch mit einem einfachen Array (oder mit einer sonstigen beliebigen linearen Datenstruktur) realisiert werden.

9.3 Schnitte beliebig orientierter Liniensegmente[48]

Befinden sich die Segmente in beliebiger Lage, so scheint die Situation schwieriger. Dennoch lässt sich das Sweepline-Prinzip anwenden: Wenn zu jedem Zeitpunkt alle „benachbarten Liniensegment" auf der Scanline betrachtet werden, wird mit Sicherheit kein Schnitt verpasst. *Benachbart* in diesem Zusammenhang heisst, dass der Schnitt aller Segmente mit der Scanline betrachtet wird, und die ScanlineDatenstruktur wieder eine Menge von Punkten darstellt. Die Haltepunkte sind nun wie zuvor die sortierten End- und Anfangspunkte der Segmente, zusätzlich werden jedoch weitere Punkte, nämlich die Schnitte selbst hinzukommen. Bei jedem Haltepunkt gibt es nun eine von den folgenden Situationen:

- Linkes Ende eines Segmentes: Das Segment wird in die Scanline-Datenstruktur eingefügt, und es werden zwei Schnitttests durchgeführt (mit beiden Nachbarn). Gibt es einen Schnitt, wird dieser in die Haltepunkt-Struktur eingefügt.
- Rechtes Ende eines Segmentes: Das Segment wird aus der Scanline-Datenstruktur entfernt und die beiden Nachbarn, die nun selbst benachbart sind, werden auf Schnitt geprüft. Gibt es einen, wird dieser wiederum in die Haltepunkt-Struktur eingefügt.
- Schnittpunkt zwischen A und B: A und B müssen vertauscht werden in der Reihenfolge in der Scanline-Datenstruktur, und zusätzlich werden zwei Schnitttest, mit den neuen Nachbarn von A und B durchgeführt. Gegebenenfalls werden diese Schnitte in die Haltepunkte-Struktur eingefügt. Zusätzlich wird nun der Schnitt berichtet. Dies geschieht erst hier, da sonst gewisse Schnitte mehrmals berichtet werden könnten.

Dieser Algorithmus benötigt also zwei Datenstrukturen:

1. **Scanline-Datenstruktur**: Hier werden die gerade aktiven Liniensegmente gespeichert, wobei diese eingefügt, gelöscht und mittels *get-neighbor* abgefragt werden können müssen. Es bietet sich also an, einen balancierten Blattsuchbaum (dessen Blätter verketten sind) zu verwenden.
2. **Haltepunkt-Datenstruktur**: In dieser Struktur werden alle Haltepunkte gespeichert, also alle Anfangs- und Endpunkte der Segmente (zu Beginn bekannt) und alle Schnittpunkte (werden laufend berechnet). Da ein Schnittpunkt mehrere Male eingefügt werden kann, genügt eine einfache Event Queue wie etwa ein binärer Heap nicht; stattdessen muss ebenfalls ein balancierter Baum verwendet werden.

Um die Laufzeit zu analysieren, macht man folgende Beobachtungen: Der Algorithmus passiert im allgemeinen 2n+k Haltepunkte. An jedem Haltepunkt wird eine konstante Zahl von Operationen ausgeführt, wobei jeweils die Grösse der jeweiligen Binärbäume über die Zeitkomplexität entscheiden. Da sich höchstens $\mathcal{O}(n + k) = \mathcal{O}(n^2)$ Elemente in den Bäumen befinden, sind alle Operationen in logarithmischer Zeit durchführbar. Damit ergeben sich folgende Gesamtlaufzeiten:

- detect: $\mathcal{O}(n \log n)$ (beim 1. Schnittpunkt abbrechen)
- report: $\mathcal{O}\big((n + k) \log n\big)$ bei k Schnitten, mit $k = \mathcal{O}(n^2)$

9.4 Schnitt achsenparalleler Rechtecke[49]

Um eine Menge von n iso-orientierter Rechtecke in der Ebene auf Schnitte zu prüfen, kann folgende Überlegung gemacht werden: Mithilfe eines Scanline-Verfahrens kann das statische Problem in der Ebene in eine Abfolge von dynamischen Intervallschnitt-Problemen überführen. Wir brauchen also eine Datenstruktur, in welche wir Intervalle einfügen und entfernen können. Zusätzlich soll effizient bestimmt werden können, welche Intervalle in der Struktur für ein gegebenes Intervall gemeinsame Punkte besitzen.

Um das Problem weiter zu vereinfachen, kann folgende Überlegung gemacht werden:

---

[48] Stefan Heule
[49] Stefan Heule

Sei $[a,b]$ ein gegebenes Intervall, welches wir auf Schnitt prüfen wollen. Dann sind $[c,d]$ alle Intervalle, welche wir berichten müssen: $\{[c,d] | [a,b] \cap [c,d] \neq \emptyset\}$. Dieser Ausdruck lässt sich nun folgendermassen umformen: $\{[c,d] | a \text{ spiesst } [c,d] \text{ auf}\} \cup \{[c,d] | c \text{ liegt im Bereich } [a,b]\}$.

Folglich genügt es, eine Datenstruktur zu finden, welche die für die gespeicherte Menge an Intervallen und ein Intervall $[a,b]$ folgende zwei Fragen beantworten kann:

1. Berichte alle Intervalle $[c,d]$, die vom linken Randpunkt a aufgespiesst werden.
2. Berichte alle Intervalle $[c,d]$, deren linker Randpunkt c im Bereich $[a,b]$ liegt.

Die zweite Frage lässt sich natürlich leicht mit einem range-tree (Bereichs-Suchbaum) beantworten, die erste Frage hingegen ist neu.

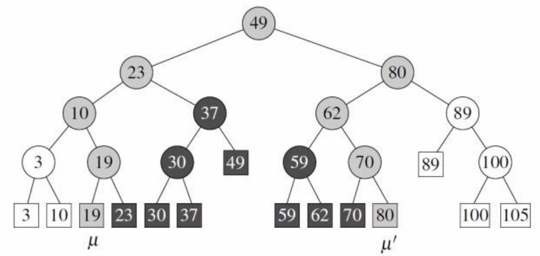
Um die aktiven Rechtecke zu verwalten, benötigt man eine Datenstruktur, die Intervalle speichern und entfernen kann, sowie die obigen Fragen beantworten kann. Wie erwähnt lässt sich die zweite Frage einfach mit einem range-tree beantworten. Für die erste Frage lässt sich ein Intervall- oder Segmentbaum einsetzen.

Insgesamt lässt sich das Problem also in $\mathcal{O}(n \log n + k)$ Zeit lösen. Je nachdem für welchen Baum man sich entscheidet, gibt dies einen unterschiedlichen Platzverbrauch: Die komplette Lösung mittels Intervaltree braucht $\mathcal{O}(n)$ Speicherplatz, während man mit einem Segmentbaum bei $\mathcal{O}(n \log n)$ Speicher liegt.

9.5 Range tree[50]

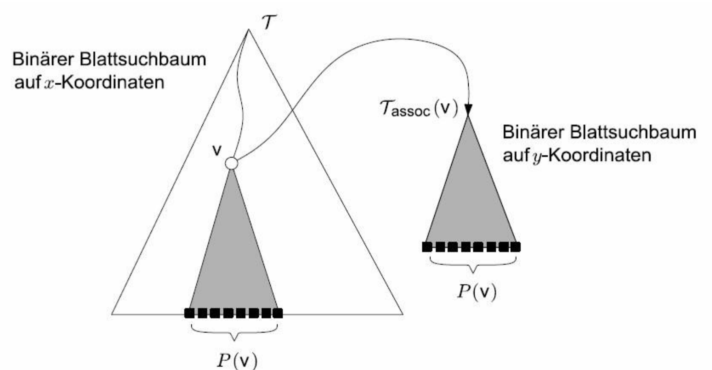
Range trees erlauben orthogonale Bereichsanfragen in einer *d*-dimensionaler Punktemenge in $\mathcal{O}(\log^d n + k)$ Zeit bei einem Platzverbrauch von $\mathcal{O}(n \log^{d-1} n)$. Verwendet werden diese Bäume beispielsweise im Rechteckschnittproblem.

9.5.1 1-dimensionaler range tree



Die 1-dimensionale Variante erlaubt auf einer Zahlenmenge eine Bereichsanfrage, die in logarithmischer Zeit bewältig werden kann. Dazu verwendet man einen Blattsuchbaum, bei dem die Blätter in einer verketteten Liste gespeichert werden. Dies hat zur Folge, dass für eine Bereichsanfrage nach dem einen Ende gesucht werden kann (das geht in logarithmischer Zeit wenn ein balancierter Baum eingesetzt wird), und dann wird einfach so lange der Liste gefolgt, bis das erste Element ausserhalb des Bereiches gefunden wird. Will man nur die Anzahl Punkte in einem Intervall wissen, so muss bei jedem inneren Knoten zusätzlich gespeichert werden, wie viele Punkte in diesem Teilbaum gespeichert sind. Dann werden einfach die Kinder aller *„ausgewählter Teilbäume"* (siehe Abschnitt 2 und Graphik) gezählt, was in logarithmischer Zeit machbar ist.

9.5.2 d-dimensionaler range tree



Die d-dimensionale Erweiterung speichert für jeden Teilbaum T_x alle Knoten in T_x in einem $(d-1)$-dimensionalen Baum. Genauer:

- Die Basis besteht aus einem balancierten binären Blattsuchbaum T, sortiert nach der ersten Koordinate der Datensätze.
- Jeder innere Knoten v besitzt einen Zeiger auf einen weiteren Blattsuchbaum $T_{assoc}(v)$. In diesem sind Kopien aller Blätter gespeichert, jedoch sortiert nach der nächsten Koordinate.

Es gibt also für jede Dimension eine Menge von Bäumen und jeder Schlüssel kommt in asymptotisch $\mathcal{O}(\log^{d-1} n)$ vielen Bäumen vor.

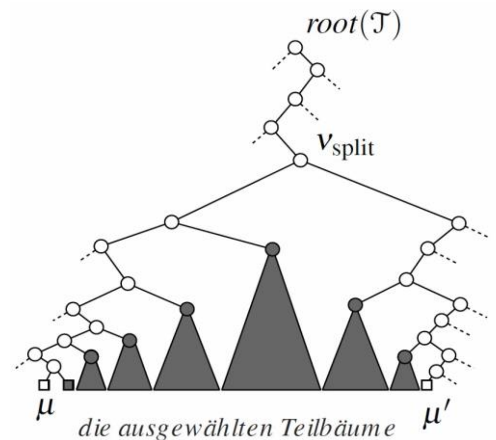
Eine Suchanfrage funktioniert nun folgendermassen: Zuerst wird nach der 1. Koordinate im Basisbaum gesucht: Dazu wird nach den beiden Enden des Anfrageintervalls $[a,b]$ von der Wurzel aus gesucht, wobei sich die beiden Suchpfade irgendwann in einem Knote v_{split} trennen. Von dort folgen wir einem Pfad nach links zu *a*, wobei wir jedes Mal, wenn

[50] Stefan Heule

wir links abbiegen, eine (d-1)-dimensionale Bereichsanfrage im rechten Kind durchführen. Der Basisfall ist dann die 1-dimensionale Bereichsanfrage wie oben beschrieben. Symmetrisch gehen wir auch für den Pfad zu *b* vor.

9.6 Segment tree[51]

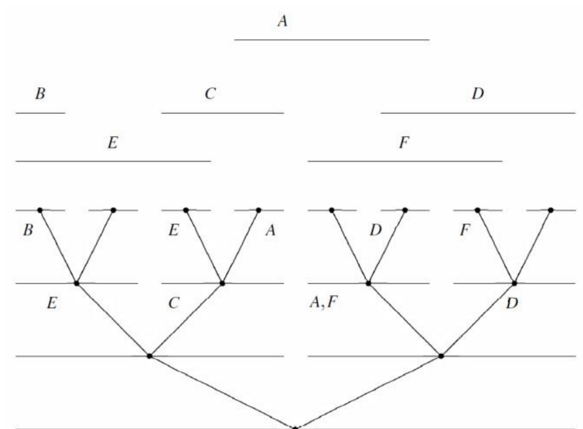
Segmentbäume sind eine halbdynamische Skelettstruktur: Zuerst wird eine leeres Skelett aufgebaut, in welches nachher dynamisch Intervalle (Segmente) eingefügt und entfernt werden können. Zudem erlaubt die Datenstruktur effiziente Verarbeitung von sogenannten Aufspiessanfragen; das heisst, für einen gegebenen Punkt, in welchen Intervallen liegt dieser?



9.6.1 Struktur und Operationen

Das Skelett besteht aus einem vollständigen Binärbaum, wobei Intervalle mit Endpunkten aus einer fixen Menge $\{1, \ldots, n\}$ aufgenommen werden können. Die Blätter stehen für elementare Intervalle $[i, i+1]$, und jeder innere Knoten im Baum repräsentiert die Vereinigung aller elementaren Intervalle in seinem Teilbaum. Die Wurzel repräsentiert also das gesamte Intervall $[1, n]$.

Ein Intervall wird nun folgendermassen im Baum gespeichert: Für jeden Knoten, der ein Intervall repräsentiert, welches vollständig im einzufügenden Intervall liegt und am nächsten bei der Wurzel liegt, wird dort der Name des Intervalls vermerkt. Mit der Überlegung, dass die Endpunkte jedes Intervalls auf zwei Pfaden die sich nur einmal trennen (vgl. range trees) erreichbar sind, lässt sich zeigen dass der Name jedes Intervalls an höchstens $O(\log n)$ Knoten vorkommt. Der Speicherbedarf ist deshalb $O(n \log n)$. Die Listen der Namen bei jedem Knoten können in einer linearen Liste verwaltet werden.



9.6.1.1 Einfügen und Löschen

Das **Einfügen** eines Intervalls $[a, b]$ ist einfach: Man beginnt bei der Wurzel und geht rekursiv vor. Bei jedem Knoten wird entschieden, ob das von ihm repräsentierte Intervall vollständig in $[a, b]$ liegt. Wenn dem so ist, wird der Name in diesem Knoten gespeichert und man ist fertig. Andernfalls wird für beide Kinder überprüft, ob es gemeinsame Punkte in $[a, b]$ und dem Intervall des Kindes gibt. Wenn dem so ist, wird die Funktion rekursiv für das Kind aufgerufen. Da höchstens logarithmisch viele Knoten betrachten werden beträgt die Laufzeit $O(\log n)$.

Beim Löschen kann man grundsätzlich genauso vorgehen. Um den Intervallnamen in der verketteten Liste jedoch zu finden und entfernen sind schlimmstenfalls linear viele Schritte nötig; eine nicht akzeptabler Aufwand. Das Problem wird mit einer Sekundärstruktur für die Intervallnamen gelöst: In einem Wörterbuch, welches Einfügen und Entfernen in logarithmischer Zeit erlaubt (z.B. ein balancierter Binärbaum) werden alle Namen der Intervalle gespeichert. Jeder Eintrag zeigt auf eine lineare Liste mit Zeigern zu jedem Vorkommen des Namens in der Grundstruktur. Damit ist auch das Entfernen in logarithmischer Zeit möglich.

9.6.1.2 Aufspiessanfragen (stabbing query)

Um eine Aufspiessanfrage für einen gegeben Punkt x zu beantwortet, kann wiederum rekursiv vorgegangen werden:

```
procedure report(p: Knoten; x: Punkt)  gebe alle Intervalle von p
aus  if (p ist Blatt) fertig;  else
  if (p hat linken Sohn s mit Intervall, das x enthält)
   report(s,x);
  if (p hat linken Sohn s mit Intervall, das x enthält)
   report(s,x);
end
```

[51] Stefan Heule

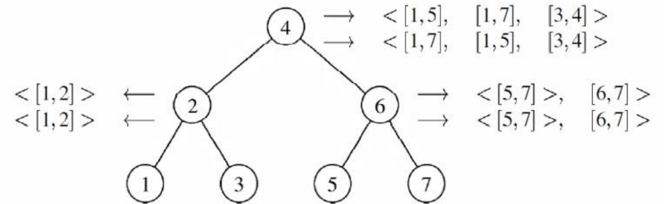
9.7 Interval tree[52]

Wie Segmentbäume dienen auch diese Bäume der Speicherung von Intervallen, und unterstützen ebenfalls Aufspiessanfragen. Diese Struktur kommt jedoch mit linearen Speicherplatz aus, verglichen mit den $\mathcal{O}(n \log n)$ der Segmentbäume.

Da eine Menge von n Intervallen höchstens s verschiedene Endpunkte haben kann, kann ohne Einschränkung die Menge $\{1, \dots, s\}$ als Menge der Endpunkte verwendet werden, wenn $s \leq 2n$.

9.7.1 Struktur und Operationen

Das Skelett besteht aus einem vollständigen Binärbaum mit den Schlüsseln $\{1, \dots, s\}$. An jedem Knoten gibt es zwei sortierte Listen *u* und *o*. In *u* sind die Intervalle aufsteigend sortiert nach den unteren Endpunkte und in *o* absteigend nach den oberen Endpunkten. Ein Intervall befindet sich nun bei nur genau einem Knoten des Baumes, nämlich beim der Wurzel am nächsten befindliche Knoten, welcher im Intervall liegt.



9.7.1.1 Einfügen und Entfernen

Für ein Intervall $[a, b]$ geschieht das Einfügen also folgendermassen: Es geschieht rekursiv und für jeden Knoten wird überprüft, ob der Schlüssel in $[a, b]$ liegt. Ist dies der Fall, wird das Intervall in die beiden Listen eingetragen, und andernfalls wird bei einem der Kinder weitergesucht: Ist a kleiner als der Schlüssel des gerade betrachteten Knotens, so wird links weitergemacht, sonst rechts. Werden die beiden Listen als balancierte Bäume organisiert, benötigt das Einfügen lediglich logarithmische Zeit.

Das Entfernen geschieht natürlich genau invers zum Einfügen und ist ebenfalls in logarithmischer Zeit machbar.

9.7.1.2 Aufspiessanfrage

Eine Anfrage, welche Intervalle von einem gegebenen Punkt *x* aufgespiesst werden, wird nun folgendermassen beantwortet: Man folgt dem Suchpfad im Baum zum Knoten mit Schlüssel x. Auf dem Weg dorthin wird bei jedem Knoten eine der beiden Listen inspiziert. Ist x kleiner als der Schlüssel des Knotens, wird die u-Liste durchgesehen und alle Intervalle ausgegeben, solange das linke Ende kleiner als x ist. Analog dazu wenn x grösser als der Schüssel ist: Die o-Liste wird durchgegangen und alle Intervalle ausgegeben, solange das rechte Ende grösser als x ist. Ist der Schlüssel gleich x, so wird eine der beiden Listen gewählt und alle Elemente ausgegeben.

Da das durchgehen der Listen in einer Zeit linear in den auszugebenden Elementen ist, dauert eine Aufspiessanfrage $\mathcal{O}(\log n + k)$.

9.7.2 Vergleich mit ähnlichen Datenstrukturen

9.7.2.1 Interval trees vs. segment trees

Beide Strukturen können Einfügen und Entfernen von Intervallen sowie Aufspiessanfragen in $\mathcal{O}(\log n)$bzw. $\mathcal{O}(\log n + k)$bearbeiten. Dennoch haben beide Strukturen verschiedene Vorteile. Für Segmentbäume gilt:

- Segmentbäume können in **beliebige Dimensionen** verallgemeinert werden, wobei sich ihre Laufzeit jeweils um einen logarithmischen Faktor verschlechter pro zusätzliche Dimension. Für Intervallbäume gibt es keine Verallgemeinerung, jedoch werden die beiden Datenstrukturen oftmals gemeinsam verwendet: Als Grundstruktur dienen Segmentbäume, und auf der letzten Stufe kommen dann Intervallbäume zum Einsatz, um etwas Speicher zu sparen.
- Die Liste der Intervalle bei jedem Knoten kann bei Segmentbäumen beliebig gespeichert werden, wodurch sie **flexibler** werden (was genau das hilft, ist mir jedoch nicht klar).

Intervallbäume haben auch verschiedene Vorteile, nämlich:

- Es wird **weniger Speicherplatz** benötigt, nämlich $\mathcal{O}(n)$ im Vergleich zu $\mathcal{O}(n \log n)$.

[52] Stefan Heule

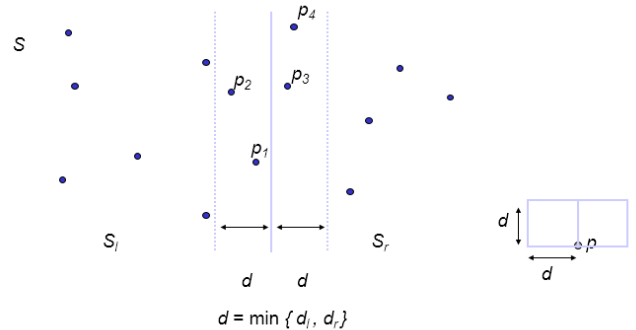
9.7.2.2 Interval and range trees

Intervallbäume sind in einem gewissen Sinne invers zu Bereichbäumen: Ist es bei ersterem möglich, Intervalle zu speichern und für einen gegebenen Punkt effizient entscheiden, welche Intervalle diesen Punkt enthalten, erlauben letztere Punkte zu speichern und für ein Intervall alle Punkte in diesem anzugeben.

9.8 Prioritätssuchbaum[53]

- Einfache Struktur für Punkte in der Ebene, platzoptimal und effizient, einfügen und entfernen in $\mathcal{O}(\log n)$
- Halbstreifenanfrage $H = [x1, x2] \times (-\infty, y]$
- Finde Punkte aus D in H
- Binärer Suchbaum für x-Koordinaten
- Min-Heap für y-Koordinaten

9.9 Geometrisches Divide-and-conquer: Paar nächster Nachbarn in Punktmenge[54]



Bestimme für eine Menge S von n Punkten ein Paar mit minimaler Distanz.

1. **Divide** Teile S in zwei gleichgroße Mengen S_l und S_r
2. **Conquer** $d_l = mindist(S_l),\ d_r = mindist(S_r)$
3. **Merge** $d_{lr} = \min\{\ d(p_l, p_r) | p_l \in S_l, p_r \in S_r\}$ $return \min\{d_l, d_r, d_{lr}\}$

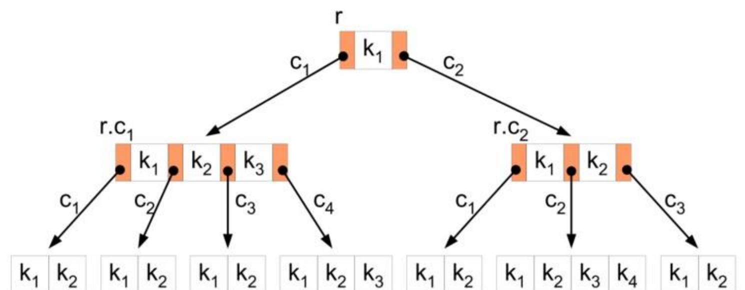
Laufzeit: $\mathcal{O}(n \log n)$

10 Externspeicher-Datenstrukturen.

10.1 Principle of Locality[55]

Nah = schneller.

10.2 B-Bäume[56]



Will man sehr grosse Mengen an Daten verwalten, reichen die einfach Datenstrukturen wie AVL-Bäume oder andere Arten von Binärbäumen nicht mehr aus. Da die Daten aus Speicherplatzmangel in sogenannten Hintergrundspeicher ausgelagert werden muss, benötigt man eine vollständig neue Sichtweise, insbesondere für die Analyse solcher Strukturen.

Der Speicher wird in *Seiten* organisiert (aufgrund des Aufbaus des Speichermediums). Es können jeweils nur eine konstante Anzahl solcher Seiten im Hauptspeicher gehalten werden, möchte man auf Daten aus einem anderen Block zugreifen, muss diese Seite erst in den RAM geladen werden. Da dieser Vorgang um mehrere Grössenordnungen langsamer ist als ein Zugriff auf normalen Hauptspeicher, will man die Anzahl Zugriffe minimieren.

10.2.1 Struktur von B-Bäumen

B-Bäume werden als Vielwegbäume realisiert: Jeder Knoten verfügt über eine bestimmte Anzahl an Schlüsseln und Zeigern, wobei sich diese Werte in bestimmten Schranken bewegen. Damit die Höhe nur logarithmisch wächst, werden an einen **B-Baum der Ordnung m** folgende strukturellen Bedingungen gestellt:

---

[53] U.a. nach http://tizian.cs.uni-bonn.de/Lehre/Vorlesungen/AlgGeom11/WH8GeomDS-PriorityAnim.pdf
[54] http://electures.informatik.uni-freiburg.de/portal/download/37/8945/Divide&Conquer.pdf; Th. Ottmann ist Co-Autor des Buches zur Vorlesung
[55] Siehe Digitaltechnik
[56] Stefan Heule

1. Alle Blätter haben die gleiche Tiefe.
2. Jeder Knoten hat mindestens $\left\lceil \frac{m}{2} \right\rceil$ Söhne. Die Wurzel ist von diese Regel ausgenommen.
3. Die Wurzel hat mindestens 2 Söhne, solange sie kein Blatt ist.
4. Jeder Knoten hat höchstens m Söhne.
5. Jeder Knoten mit i Söhnen hat i-1 Schlüssel.

Diese Bedingungen genügen, um zu garantieren, dass ein Baum höchstens eine in N logarithmische Höhe erreicht.

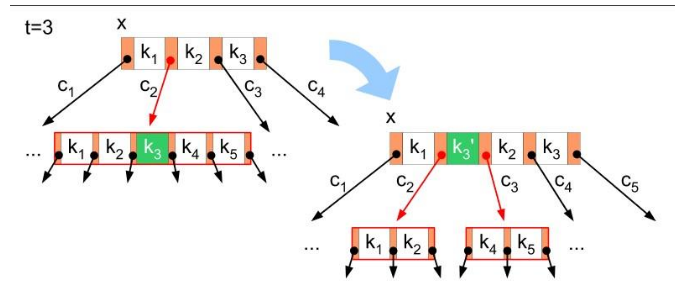
Die Zahl m hängt in der Praxis vom eingesetzten Speichermedium und kann von 100 bis zu mehreren Tausend gehen. Entscheidend ist, wie viel Schlüssel (inklusive Daten) in einer Seite gespeichert werden können, denn typischerweise wird jeweils ein Knoten pro Seite gespeichert.

10.2.2 Die elementaren Operationen auf B-Bäumen

10.2.2.1 Suchen

Beim Suchalgorithmus in B-Bäumen handelt es sich um die natürliche Erweiterung des Suchens bei Binärbäumen. Man beginnt bei der Wurzel und überprüft dann, ob sich der zu suchende Schlüssel im gerade betrachteten Knoten befindet. Ist dies nicht so und der Knoten ist kein Blatt, so wählen wir den Zeiger zwischen den beiden Schlüsseln, die denen der gesuchte Schlüssel liegt.

10.2.2.2 Einfügen

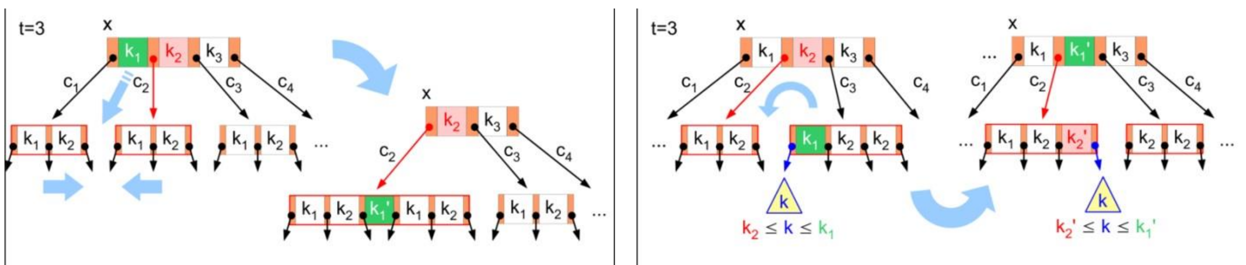


Es wird mit einer (erfolglosen) Suche nach dem einzufügenden Schlüssel x begonnen. Diese endet in einem Blatt, an der zu erwartenden Position für x. Hat der gerade betrachtete Knoten weniger als m-1 Schlüssel, können wir den Schlüssel ganz einfach als neuen Schlüssel in diesen Knoten einfügen. Ist der Knoten jedoch bereits voll, müssen wir ihn teilen. Dazu nehmen wir den Median aller Schlüssel in diesem Knoten und verschieben diesen in den Vaterknoten. Das ergibt eine weitere Unterteilung beim Elternknoten, der jetzt ein Kind mehr hat. Aus dem Kindknoten mit m-1 Schlüsseln wurde nun zwei Knoten mit je m/2 Knoten (Bedingung erfüllt), wobei in einem der beiden nun auch unser Schlüssel x gespeichert werden kann. Das Einfügen eines weiteren Schlüssels im Vaterknoten muss natürlich nicht unbedingt möglich sein, auch diese Knoten kann bereits voll sein. Dann wird einfach ein weiteres Mal geteilt, notfalls bis zur Wurzel. Ist die Wurzel auch voll, erstellt man eine neue, leere Wurzel, die dann nur eine Schlüssel enthält und damit zwei Kinder hat (Bedingung erfüllt).

Ein B-Baum wächst also nicht nach unten, sondern an der Wurzel. Damit ist auch leicht zu sehen, dass die Bedingung (1) erfüllt ist.

10.2.2.3 Löschen

Ähnlich wie beim Einfügen, wo man sich vor zu stark besetzten Knoten schützen musste, kann es hier eine Art Unterlauf geben, sodass ein Knoten weniger als die $\left\lceil \frac{m}{2} \right\rceil$ Kinder hat. Zudem ist es hier möglich, dass auch innere Knoten bearbeitet werden, was die Sache ein klein wenig komplizierter macht.



Verschmelzen zweier Knoten

Verschieben eines Schlüssels

Das Vorgehen ist aber nicht grundsätzlich verschieden: Der Schlüssel x wird zuerst gesucht. Handelt es sich dabei um einen Schlüssel eines inneren Knotens, wird der symmetrische Nachfolger von x mit x ausgetauscht, und anstelle dessen der Nachfolger gelöscht. Kann man diesen Schlüssel nun ohne Probleme löschen, sind wir fertig. Entsteht aber ein Unterlauf, so muss etwas angepasst werden. Als erstes werden die beiden Brüder des Knotens mit dem Schlüssel x, nennen wir ihn n, angesehen. Hat einer von diesen beiden mehr als die minimale Anzahl Schlüssel, können Schlüssel

von diesem Knoten transferiert werden (siehe Graphik). Ist dies bei beiden nicht der Fall, müssen n und irgendein Nachbar verschmolzen werden. Das geht, da zuvor beide $\left\lceil\frac{m}{2}\right\rceil - 1$ Schlüssel hatten, und nun n einen Schlüssel verlor. Damit hat der entstehende Knoten nach der Verschmelzung höchstens m-1 Schlüssel.

11 Additional Wisdom

- $\log n! \in \Theta(n \log n)$
- (Bei P. Widmayer) Die Sondierfunktion $s(j,k)$ wird von der Hashfunktion abgezogen
- Double-Hashing ist immer vom Startpunkt aus
- Ein Heap entspricht einer priority queue
- Allgemeine Struktur für DP
    1. Definition der DP-Tabelle (Grösse, Bedeutung eines Eintrages)
    2. Berechnung eines Eintrages
    3. Berechnungsreihenfolge
    4. Auslesen der Lösung
    5. Laufzeitanalyse
- Master-Theorem[57]

[57] http://people.csail.mit.edu/thies/6.046-web/master.pdf
http://www.math.dartmouth.edu/archive/m19w03/public_html/Section5-2.pdf
http://de.wikipedia.org/wiki/Master-Theorem
http://en.wikipedia.org/wiki/Master_theorem