Datenstrukturen
Zusammenfassung und Fragezettel

Michael Gregorius

15. März 2001

Vorwort

Diese Zusammenfassung wurde in Vorbereitung auf die sogenannte INF2-Prüfung geschrieben. Sie ist in Frage/Antwort-Form geschrieben, da eine Prüfung ja auch größtenteils nach diesem Schema abläuft. Als Hauptquelle diente das Skript *„Datenstrukturen WS 1996/97“* von *Ingo Wegener*. Teilweise sind Kapitel unvollständig. Dies ist oft der Fall, wenn der Stoff nicht wirklich prüfungsrelevant ist. Andere Kapitel aus dem Skript, wie zum Beispiel *Schnelle Fourier-Transformation* sind erst garnicht vorhanden, da es sehr unwahrscheinlich ist, daß sie in einer Prüfung abgefragt werden.

Desweiteren ist noch anzumerken, daß das Beispiel zum Thema *Dynamische Programmierung* scheinbar falsch ist. Jeder, der den Algorithmus jedoch schonmal selber von Hand durchgerechnet hat, weiß warum ich das Beispiel nicht korrigiert habe. Es ist eh wichtiger das Prinzip zu verstehen. Man überläßt solche Aufgabe ja auch nicht umsonst dem Computer.

Zuletzt kann ich nur empfehlen, zusammen mit anderen Kommilitonen zu lernen, da man dadurch lernt, den Stoff in eigenen Worten wiederzugeben.

Inhaltsverzeichnis

1 Grundlegende Datenstrukturen

1.1 Arrays

- **Was ist ein Array?**

Ein Array ist eine statische Datenstruktur, die den Zugriff auf ihre Elemente in $O(1)$ gestattet.

- **Was muß man beachten, da ein Array eine statische Datenstruktur ist?**

Man muß beachten, daß bei der Initialisierung schon feststeht, wieviele Elemente die Datenstruktur enthält und sich diese Menge auch nicht verändert.

- **Wie sieht die konkrete Realisierung im Rechner aus?**

Das Array ist ein fortlaufender Speicherbereich im Rechner, welcher bei einer bestimmten Adresse, der Startadresse N, beginnt. Die einzelnen Elemente werden über eine Speicherabbildungsfunktion bestimmt. Wenn das Element n Elemente hält, so steht das i-te Element an der Stelle $N+i$, so daß sich das Array von $N+1$ über $N+i$ bis zu $N+n$ erstreckt.

- **In welcher Zeit ist das Vertauschen von zwei Objekten möglich? In welcher Zeit das Einfügen eines Elementes zwischen zwei andere?**

Das Vertauschen von zwei Objekten ist in Zeit $O(1)$ möglich, also in konstanter Zeit. Wenn man ein Element aus seiner Position $a[i]$ nimmt und dieses zwischen zwei andere $a[j], a[j+1]$ einfügen möchte, so kann dies Zeit $\Omega(n)$ dauern, da die anderen Element entsprechend verschoben werden müssen. Wenn z.B. $i > j+1$ ist, müssen die Elemente $a[j+1], ..., a[i-1]$ verschoben werden.

- **Wie sieht die Speicherabbildungsfunktion für zweidimensionale Arrays aus?**

Die Position (i, j) in einem zweidimensionalen Array $[1...n_1] \times [1...n_2]$ sieht wie folgt aus:

$$N + (i-1)n_2 + j$$

N gibt die Startadresse an, von der aus gerechnet wird. Über $(i-1)n_2$ gibt man an, in welcher Zeile man landen möchte, wieviele komplette Zeilen man also „überfliegen" möchte. Und j sagt, welches Element man aus dieser Zeile haben möchte.

- **Wie sieht die Speicherabbildungsfunktion für ein k-dimensionales Array aus?**

Für ein k-dimensionales Array der Form $[1...n_1] \times [1...n_2] \times ... \times [1...n_k]$ sieht die Speicherabbildungsfunktion wie folgt aus:

$$N + \sum_{j=1}^{k-1} \left((i_j - 1) \prod_{l=j+1}^{k} n_l \right) + i_k$$

Sie erklärt sich, wie folgt: N gibt auch hier die obligatorische Startadresse an, von der aus gerechnet wird. $\sum_{j=1}^{k-1} \left((i_j - 1) \prod_{l=j+1}^{k} n_l \right)$ gibt an, wie viele Superzeilen übersprungen werden sollen. Dabei bestimmt $\prod_{l=j+1}^{k} n_l$ wie groß eine Superzeile ist und $(i_j - 1)$, wie viele davon übersprungen werden sollen. Am besten ein Beispiel:

Ein dreidimensionales Array soll abgespeichert werden. Wenn ein eindimensionales Array eine Linie ist und ein zweidimensionales Array eine Fläche ist, so kann man sich ein dreidimensionales Array als einen Würfel vorstellen, bzw. noch besser, als eine Aufschichtung von Flächenarrays.

Angenommen, wir haben ein $[1...5] \times [1...4] \times [1...3]$-Array. Es wäre insgesamt also 60 Einträge groß. Wir wollen den Eintrag [3,1,2] bestimmen. Stellen wir uns das Array als ein Hochhaus vor, so müssen wir also in die dritte Etage. Dafür müssen wir zwei komplette Etagen überspringen. Diese „Etagen" sind $4 \cdot 3 = 12$ Einzelfelder groß. Damit ergibt sich schonmal die Formel:

$$N + (3-1) \cdot 4 \cdot 3 + ... = N + 2 \cdot 4 \cdot 3 + ... = N + 24 + ...$$

Damit haben wir schon die richtige Etage gefunden. Jetzt müssen wir nur noch sagen, wohin wir in dieser Etage wollen. Dazu spezifizieren wir erst mal die Zeile, in die wir möchten. Dies geschieht in unserem Beispiel durch $j = 2$ in der Summe. Es ergibt sich:

$$... + (1-1) \cdot n_3 + ... = ... + 0 \cdot 3 + ... = ... + 0 + ...$$

Jetzt müssen wir nur noch bestimmen, welches Element wir aus dieser Zeile haben möchten. Dies geschieht über $i_k : ... + 2$.

Insgesamt ist das Element $[3,1,2]$ also an Position $N + 24 + 0 + 2 = N + 26$, also an der 26. Position von der Startadresse aus.

Allgemein sieht die Formel für ein dreidimensionales Array wie folgt aus:

$$N + (i_1 - 1) \cdot n_2 \cdot n_3 + (i_2 - 1) \cdot n_3 + i_3 = N + \sum_{j=1}^{2} \left((i_j - 1) \prod_{l=j+1}^{3} n_l \right) + i_3$$

In einem vierdimensionalen Array würde das ganze noch eine „Ebene" höher gehen. Ein vierdimensionales Array ist ein Hyper-Cubus und um die Position zu bestimmen, müssen wir von der Startadresse N aus bestimmen, in welchen Würfel wir wollen, in welche Etage dieses Würfels, in welche Zeile die Etage und das wievielte Element aus dieser Zeile.

- **Wie ist eine untere Dreiecksmatrix definiert?**

Eine Matrix heißt untere Dreiecksmatrix, wenn $M(i, j) = 0$ für $i < j$ ist. Ein Beispiel für eine untere Dreiecksmatrix:

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Wenn der Zeilenindex also kleiner als der Spaltenindex ist, wird das entsprechende Element 0 gesetzt.

- **Wieviele Elemente hat eine untere Dreiecksmatrix, von der nur die Elemente die nicht trivial** 0 **sind in einem Array abgespeichert werden?**

Auch hier geht wieder mal der kleine Gauss ein. Angenommen, wir haben eine untere $n \times n$-Dreiecksmatrix. In der ersten Zeile steht ein Element, welches nicht trivial 0 ist ($a_{11} \neq 0$). In der zweiten Zeile sind dies zwei, in der i-ten Zeile i und dementsprechend in der n-ten n. Die Gesamtzahl ergibt sich also aus:

$$\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$$

Man spart fast 50%, denn es gilt: $\frac{\frac{1}{2}n(n+1)}{n^2} = \frac{\frac{1}{2}n^2+\frac{1}{2}n}{n^2} = \frac{n^2}{2n^2} + \frac{1}{2n} = \frac{1}{2} + \frac{1}{2n}$.
Weiter gilt: $\lim_{n\to\infty}\left(\frac{1}{2}+\frac{1}{2n}\right) = \lim_{n\to\infty}\frac{1}{2} + \lim_{n\to\infty}\frac{1}{2n} = \frac{1}{2}+0 = \frac{1}{2}$. Mit steigendem n spart man also immer mehr Platz ein.

- **Wie sieht die Speicherabbildungsfunktion für eine untere Dreiecksmatrix aus?**

$M(i,j)$ steht für $i \geq j$ an folgender Position:

$$N + (1 + ... + (i-1)) + j = N + \left(\sum_{k=1}^{i-1} k\right) + j = N + \frac{1}{2}(i-1)i + j$$

Ähnlich wie bei den k-dimensionalen Arrays gibt auch hier N die Startadresse an. Über $\frac{1}{2}(i-1)i$ gibt man die Zeile an, in die man möchte und j definiert einfach die Spalte.

- **Was ist eine m-Bandmatrix?**

Eine Matrix M heißt m-Bandmatrix, wenn $M(i,j) = 0$ für alle $|i-j| \geq m$ gilt.

Beispiel:

Wir betrachten eine 9×9-Matrix, die eine 3-Bandmatrix ist. Auch hier gilt $n > 2m$, damit sich die besondere Abspeicherung überhaupt lohnt. Die Matrix sieht wie folgt aus:

$$\begin{pmatrix}
a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 & 0 & 0 \\
a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 & 0 & 0 \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 & 0 & 0 \\
0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 & 0 & 0 \\
0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & 0 & 0 \\
0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} & 0 \\
0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} & a_{78} & a_{79} \\
0 & 0 & 0 & 0 & 0 & a_{86} & a_{87} & a_{88} & a_{89} \\
0 & 0 & 0 & 0 & 0 & 0 & a_{97} & a_{98} & a_{99}
\end{pmatrix}$$

Wie man sieht, fängt diese Matrix mit 3 [m] Elementen in der ersten Zeile an und wächst bis auf 5 [$2m-1$] Elemente in der dritten [m-ten] Zeile an. In den letzten 3 [m] Zeilen fällt sie entsprechend von 5 [$2m-1$] auf 3 [m] Zeilen. Der Fall fängt also bei Zeile 7 [$n-m+1$] an.

Allgemein kann man also drei Fälle unterscheiden:

1. Das gesuchte Element befindet sich in einem Bereich, in dem die Anzahl der Einträge pro Zeile noch ansteigt. Im Beispiel Zeile 1 – 3.

2. Das gesuchte Element befindet sich in einem Bereich, in dem die Anzahl der Einträge pro Zeile konstant bleibt. Im Beispiel Zeile 4 – 6.

3. Das gesuchte Element befindet sich in einem Bereich, in dem die Anzahl der Einträge pro Zeile wieder stagniert. Im Beispiel Zeile 7 – 9.

- **Wie sieht die Speicherabbildungsfunktion für die drei Fälle aus?**

Wie schon oben beschrieben, gibt es drei Fälle, zwischen denen man differenzieren muß.

Fall 1: Wir befinden uns in einer der ersten m Zeilen. Die Anzahl der Elemente pro Zeile ist also nicht konstant und es gilt: $i \leq m$.

Wie man sieht sind in diesem Bereich in jeder Zeile mindestens m Einträge, die nicht 0 sind. Und von einer Zeile zur nächsten wächst die Anzahl der Nicht-0-Einträge jeweils um 1. Die Speicherabbildungsfunktion geht also ans Ende einer Zeile und zählt dann noch j ab. Man kann die aktuelle Position also berechnen, indem man angibt, wieviele m-Zeilen man überspringt, plus die Anzahl um die die Einträge bis zur i-ten Zeile noch angewachsen sind. Dies sieht dann so aus:

$N + \underbrace{(i-1)m}_{\#m} + (0+1+...+(i-2)) + j = N + (i-1)m + \left(\sum_{k=0}^{i-2} k\right) + j = N +$
$(i-1)m + \frac{1}{2}(i-2)(i-1) + j$

Fall 2: Wir befinden uns zwischen der m-ten und der $(n-m+1)$-ten Zeile, das heißt: $m < i < (n-m+1)$

Als erstes müssen wir den Bereich überspringen, in dem die Anzahl der Einträge pro Zeile ansteigt, also die ersten m Zeilen. Die geschlossene Form um die Anzahl der Einträge der ersten m Zeilen zu berechnen, sie wie folgt aus:

$$\begin{aligned} m \cdot m + 0 + 1 + ... + m - 1 &= m^2 + \sum_{i=0}^{m-1} i \\ &= m^2 + \frac{1}{2}(m-1)m \\ &= m^2 + \frac{1}{2}m^2 - \frac{1}{2}m \\ &= \frac{3}{2}m^2 - \frac{1}{2}m \end{aligned}$$

Also steht $M(i,j)$ an Position:

$N + \left(\frac{3}{2}m^2 - \frac{1}{2}m\right) + (i-m-1)(2m-1) + j - (i-m)$
$= N - \frac{1}{2}m^2 - \frac{1}{2}m + 2mi - 2i + j + 1$

Dabei gibt $(i-m-1)(2m-1)$ die Anzahl der komplett mit $2m-1$ Einträgen besetzten Zeilen an, die übersprungen werden sollen. Über $j-(i-m)$ findet man dann die richtige Speicherzelle. Da pro Zeile nur $2m-1$ Elemente im Array gespeichert sind, j jedoch größer als $2m-1$ sein kann, muß man dies über $(i-m)$ korrigieren.

Möchte man zum Beispiel den Eintrag $M(m+1,2)$ haben, rutscht man in die erste Zeile und über $2-(m+1-m) = 1$ ergibt sich, daß dies der erste Eintrag aus dem Band ist.

Fall 3: Der gesuchte Eintrag steht in dem Teil der Matrix, in dem die Einträge von einer Zeile zur anderen stagnieren. Dies gilt ab der $(n-m+1)$-ten Zeile. Es gilt also: $i \geq (n-m+1)$

- **Was ist eine spärlich besetzte Matrix?**

Eine spärlich besetzte Matrix ist eine Matrix, bei der nur sehr wenige Einträge von 0 verschieden sind. Ist i die Anzahl der Einträge einer $n \times m$-Matrix, gilt also $i \ll n \cdot m$.

- **Welche Möglichkeiten gibt es, um spärlich besetzte Matrizen zu speichern?**

Die Koordinatenmethode und die Bitmusterdarstellung.

- **Wie funktioniert die Koordinatenmethode?**

Bei der Koordinatenmethode wird jedes Matrixelement als ein Tripel aus Zeilenindex, Spaltenindex und Datum abgespeichert. Also in der Form $(i, j, M(i, j))$. Bei n von Null verschiedenen Elementen, genügt also ein Array der Länge n mit den Tripeln als Inhalt. Auch hier muß man beachten, daß ein Array eine statische Datenstruktur ist, also die Anzahl der von Null verschiedenen Elemente bekannt sein sollte.

Die Tripel werden dabei lexikographisch sortiert, das heißt:

$$(i, j) < (i', j') :\Leftrightarrow i < i' \vee (i = i' \wedge j < j')$$

Wenn man nun $M(i, j)$ ermitteln möchte, so sucht man im Array entweder bis man das Tripel $(i, j, M(i, j))$ findet oder aber bis man zwei benachbarte Tripel (i', j') und (i'', j'') findet für die gilt: $(i', j') < (i, j) < (i'', j'')$.

Im ersten Fall steht der Wert $M(i, j)$ an der dritten Position des Tripels und sonst ist $M(i, j) = 0$.

Dabei kann im Array entweder per linearer oder binärer Suche gesucht werden.

- **Wie funktioniert die Bitmusterdarstellung?**

Bei der Bitmusterdarstellung macht man davon Gebrauch, daß man in einem Maschinenwort der Länge l auch l Bits abspeichern kann. Eine Matrix, deren Einträge nur aus Nullen oder Einsen besteht, kann also wesentlich effizienter abgespeichert werden.

Sei M^* eine solche Matrix. Weiter sei $M^*(i, j) = 0$, falls $M(i, j) = 0$ und sonst $M^*(i, j) = 1$. Zur Berechnung von $M(i, j)$ wird zuerst $M^*(i, j)$ angeschaut. Ist $M^*(i, j) = 0$, so ist auch $M(i, j) = 0$. Sonst zählt man die Anzahl z der Elemente für die gilt $(i', j') < (i, j) \wedge M^*(i', j') = 1$. z ist also die Anzahl aller Elemente die lexikographisch vor (i, j) stehen und die Eins sind. Danach schaut man in einem Array A, welches die Werte der Positionen speichert, die ungleich Null sind, an der Position $z + 1$ nach und bekommt so $M(i, j)$.

Beispiel:

$$M = \begin{pmatrix} 0 & 4 & 0 \\ 0 & 0 & 22 \\ -3 & 0 & 0 \end{pmatrix} :\Leftrightarrow M^* = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \wedge A := \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 22 & -3 \\ \hline \end{array}$$

Wir wollen das Element $(2, 3)$ finden. $M^*(2, 3) = 1$ und es gibt $z = 1$ Elemente die lexikographisch vor $(2, 3)$ stehen und für die $M^*(i, j) = 1$ ist. Also steht $M(2, 3)$ im Array A an Position $z + 1 = 2$. Tatsächlich ist $A[2] = 22$.

- **Was muß man beachten, wenn man Queues und Stacks mit Hilfe eines Arrays implementiert?**

Auch hier muß man beachten, daß ein Array eine statische Datenstruktur ist und es dementsprechend eine Obergrenze n gibt, die nicht überschritten werden darf. Mit anderen Worten, die Queue (bzw. das Stack) kann nur n Elemente aufnehmen. Ansonsten benötigt man Routinen zur Overflowerkennung.

- **Was ist ein Stack und welche Operation unterstützt es in welcher Zeit?**

Ein Stack ist eine Folge von Objekten $a_1, ..., a_i$, die wir uns als Stapel mit a_i als Topelement und a_1 als Bottomelement vorstellen. Wie bei einem richtigen Stapel kann man nur das oberste Element herunternehmen und neue Elemente, die eingefügt werden, werden immer nur oben eingefügt. Ein Stack ist also eine *LIFO*-Datenstruktur (*last in-first out*).

Folgende Operationen sind in Zeit $O(1)$ ausführbar:

1. *EMPTY(S):* Es soll ein leerer Stack mit Namen *S* initialisiert werden.
2. *TOP(S):* Es wird das Topelement vom Stack *S* berechnet. Also das Element, welches zuletzt auf den Stack gelegt wurde. Bei einem leeren Stack *S* wird *NIL* zurückgegeben.
3. *POP(S):* Das Topelement wird von *S* entfernt. Bei einem leeren Stack *S* wird eine Fehlermeldung ausgegeben.
4. *PUSH(a,S):* Das Objekt *a* wird auf den Stapel *S* gelegt und ist somit neues Topelement.

- **Wie wird ein Stack konkret mit Hilfe eines Arrays implementiert?**

Ein Stack mit n Objekten kann in mit einem Array der Länge n und einer Extravariablen *top* realisiert werden. Dabei zeigt *top* jeweils auf die Position des Topelementes.

- **Was ist eine Queue?**

Eine Queue ist eine Folge von Objekten $a_1, ..., a_i$, die wir uns als eine Warteschlange vorstellen. Dabei ist a_1 das Kopfelement (*head*) und a_i das Endelement (*tail*). Stets wird das Kopfelement entnommen und hinter das Endelement eingefügt. Eine Queue ist also eine *FIFO*-Datenstruktur (*first in-first out*).

- **Welche Operationen unterstützt eine Queue in welcher Zeit?**

Eine Queue unterstützt folgende drei Operationen in Zeit $O(1)$:

1. *EMPTY(Q):* Erzeugt eine leere Queue mit Namen *Q*.
2. *REMOVE(Q):* Bestimmt und entfernt das Kopfelement. Das folgende Element wird dabei neues Kopfelement. Bei einer leeren Queue wird eine Fehlermeldung ausgegeben.
3. *ENTER(a,Q):* Das Objekt *a* wird neues Endelement von *Q*.

- **Wie wird eine Queue mit Hilfe eines Arrays realisiert?**

Für eine Queue, welche höchstens n Elemente halten kann, benötigt man ein Array der Länge n, sowie zwei Extravariablen *head* und *tail*, welche jeweils die Position des Kopf- bzw. Endelementes speichern. Bei einer leeren Queue zeigen *head* und *tail* beide auf *NIL*. Wird ein Element in eine leere Queue eingefügt, so wird es an der ersten Arrayposition abgelegt.

- **Eine Queue wächst hinten und schrumpft vorne. Was kann man machen, um zu verhindern, daß alle Arrayelemente nach einem REMOVE um einen Schritt nach vorne gerückt werden müssen?**

Man legt das Array ringförmig an. Im Falle, daß die Queue im Array gerade von i bis n geht und man einen *ENTER*-Befehl ausführen möchte, steht man vor einem Problem. Die Array-Position $n+1$ gibt es nicht. Wenn jedoch die erste Position frei ist (also $i \neq 1$), kann man diese nutzen. In der Praxis realisiert man dies mit einer einfachen Modulo-Rechnung. Also:

```
int Array[n];
int head, tail;
```

```
Array[(tail + 1) mod n] = Neuer Wert;
tail = (tail + 1) mod n
```

Es kann also *head* > *tail* gelten.

1.2 Lineare Listen

- **Was ist eine lineare Liste?**
Eine geordnete Folge von Objekten $a_1 ... a_n$, mit der Länge n.
- **Was ist der Vorgänger bzw. Nachfolger von a_i?**
a_{i+1} ist der Nachfolger von a_i, wenn $i \leq n$. Der Nachfolger von a_n ist *NULL*. Dementsprechend ist a_i der Vorgänger von a_{i+1}.
- **Welche Operationen unterstützt eine lineare Liste?**

1. *INSERT(x,p,L):* Füge das Objekt x in L vor a_p ein. Ist $p = n+1$, so füge x hinter a_n ein.
2. *DELETE(p,L):* Lösche das Objekt x_p aus Liste L.
3. *FIND(x,L):* Gib die erste Position p an der das Objekt x in L steht.
4. *SEARCH(p,L):* Gib das Objekt x_p, welches an Position p in L steht.
5. *PRED(p,L):* Bestimme den Vorgänger des Objektes x_p in L.
6. *SUCC(p,L):* Bestimme den Nachfolger des Objektes x_p in L.
7. *NULL(L):* Schaffe eine leere Liste mit Namen L.
8. *FIRST(L):* Gib das erste Objekt x_1 aus L.
9. *LAST(L):* Gib das letzte Objekt x_n aus L.
10. *LENGTH(L):* Berechne die Länge von L.
11. *UNION(L_1, L_2, L):* Erzeuge aus den Listen $L_1 = \{a_1, ..., a_n\}$ und $L_2 = \{b_1, ..., b_m\}$ die Liste $L = \{a_1, ..., a_n, b_1, ..., b_n\}$.
12. *COPY(L,L'):* Lege in L' eine Kopie von L an.
13. *SPLIT(p,L):* Falls $L = \{a_1, ..., a_n\}$, zerlege L in $L_1 = \{a_1, ..., a_p\}$ und $L_2 = \{a_{p+1}, ..., a_n\}$.

- **Wie realisiert man in Programmiersprachen Listen?**
Mit Hilfe von Zeigertypen. Ein Listenelement hat einen Inhalt, sowie einen Zeiger auf das nächste Element. Das letzte Element zeigt dabei auf *NULL*.
- **Was ist eine vollständige (partielle) Ordnung?**
Eine Ordnung ist vollständig, wenn (1)-(4) gilt, und partiell, wenn (2)-(4) gilt.

(1) $\forall x, y \in M : x \leq y \vee y \leq x$

(2) $\forall x \in M : x \leq x$ (Reflexivität)

(3) $\forall x, y \in M : x \leq y \wedge y \leq x \Rightarrow x = y$. (Symmetrie)

(4) $\forall x, y, z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z$ (Transitivität)

• **Nenne Beispiele für partielle Ordnungen, die nicht vollständige Ordnungen sind!**
Potenzmengen einer endlichen Menge sind bezüglich der Teilmengenrelation $\subseteq$ partiell geordnet, nicht jedoch vollständig. So gilt zum Beispiel für $M = \{1,2,3\}$:
$\mathcal{P}(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
$\{1\} \subseteq \{1,2\} \subseteq \{1,2,3\}$ sowie $\{2\} \subseteq \{2,3\} \subseteq \{1,2,3\}$
Was ist jedoch mit $\{1,2\}$ und $\{2,3\}$? Diese Elemente lassen sich nicht vergleichen, d.h (1) gilt für sie nicht.

• **Zeige Lemma 2.3.2: Jede partiell geordnete endliche Menge M lässt sich topologisch sortieren.**
Der Beweis erfolgt über den Nachweis der Existenz eines minimalen Elementen $z \in M$. D.h. es gibt kein $y \in M : y \neq z \wedge y \leq z$. Dabei muss z nicht eindeutig sein. In $\mathcal{P}(\{1,2,3\})$ erfüllen zum Beispiel $\{1\}, \{2\}$ und $\{3\}$ diese Bedingung, nachdem $\emptyset$ aus der Menge entfernt wurde. Also nimmt man ein minimales z, setzt $m_1 = z$ und iteriert das Verfahren mit $M - \{z\}$.

Der Nachweis, dass ein solches minimales Element tatsächlich in jeder endlichen Menge existiert erfolgt über einen Widerspruchsbeweis:

Man wähle ein $y \in M$. Entweder ist es minimal oder es exisitiert ein anderes Element ungleich y, welches kleiner ist, d.h.: $d(y) \neq y \wedge d(y) \leq y$. Wenn M kein minimales Element hat, was wir ja annehmen, ist $d(y)$ für alle $y \in M$ wohldefiniert, da es für jedes Element ein kleineres Element geben muss.
Also bilden wir für ein beliebiges $y \in M$ die Kette:

$$y, d(y), d^2(y) := d(d(y)), ..., d^n(y)$$

Diese Kette hat jedoch $n+1$ Elemente, während M nur n Elemente hat, d.h. zwei dieser Elemente müssen identisch sein: $\exists i, j : 0 \leq i < j \leq n : d^i(y) = d^j(y)$. Wir benennen $d^i(y)$ der Übersicht halber in x um, also $x = d^i(y)$. Weiter gelte $k = j - i$, d.h. k ist die Anzahl der Auswahlen anderer Elemente, die ich treffe, bis ich wieder bei x bin. Dann gilt $x = d^k(x)$. Nach k Schritten wähle ich x wieder aus. Nach Definition von d ist $k \geq 1$ und aufgrund der Transitivität gilt:

$$x = d^k(x) \leq d(x) \leq x \Rightarrow x = d(x)$$

$x = d(x)$ ist jedoch ein Widerspruch zur Annahme.

• **Erkläre den naiven Algorithmus!**
Gegeben seien Paare der Form (i, j), wobei dies bedeutet, daß $x_i \leq x_j$ ist. Weiter gelte $1 \leq i \leq n$, sowie $1 \leq j \leq n$. Also: $(i, j) \Rightarrow x_i \leq x_j$. Dabei wird auf triviale Paare wie (i,i) verzichtet, sowie auf Paare, die sich aufgrund der Transitivität ergeben. Das heißt, wenn gilt: (i, j) und (j,k), so wird das Paar (i,k) nicht mehr aufgeführt.

Der naive Algorithmus durchläuft die Paare und schaut nach, für welches j es kein Paar $(\cdot, j)$ gibt. Diese j sind minimal und können in die Ausgabe geschrieben werden. Danach werden alle Paare $(j, \cdot)$ gelöscht und das Spiel beginnt von vorne, bis keine Paare mehr übrig sind. Die j, welche schon in der Ausgabe stehen, werden dabei natürlich auch nicht mehr berücksichtigt. Sind keine Paare mehr übrig, so werden die restlichen j, die eventuell noch nicht in der Ausgabe stehen auch noch ausgegeben.

• **Gebe ein Beispiel an, für das der naive Algorithmus eine Laufzeit von $\Theta(n^2 + np)$ hat!**

p sei die Anzahl der Paare. Für $p = n-1$ hat der Algorithmus eine Laufzeit von $\Theta(n^2+np)$. Hierbei n der Indexbereich, d.h. die Indizes laufen von 1 bis n: $x_1,...,x_n$.

Wir benutzen zwei Arrays N und P. N speichert n boolesche Variablen und P speichert die Paare. P ist somit p Elemente lang. Wir durchlaufen das Array P und für jedes Paar (i,j), welches wir finden, setzen wir $N(j) = 1$. Nachdem P durchlaufen ist, durchlaufen wir das Array N und schreiben jedes j für das gilt $N(j) = 0$ in die Ausgabe. Danach durchläuft man wieder das Array P und schaut nach, für welche j, die noch nicht in der Ausgabe stehen, es nun keine Paare $(\cdot, j)$ mehr gibt. Dieses Spielchen wiederholt sich, bis alle Einträge in N Null sind.

Für den Fall $p = n-1$ finden wir mit dieser Methode immer nur ein neues j, welches noch nicht in der Ausgabe steht und für welches es kein Paar $(\cdot, j)$ gibt. Wir durchlaufen also n-mal das P-Array ($\Theta(np)$) und durchlaufen dementsprechend auch n-mal das Array N ($\Theta(n^2)$). Daraus ergibt sich eine Gesamtzeit von $\Theta(n^2+np)$.

- **Gebe einen Algorithmus an, der eine Laufzeit von $O(n+p)$ hat!**

Mit Hilfe von linearen Listen kann man die Laufzeit auf $O(n+p)$ drücken. Dazu brauchen wir ein Array A der Länge n. Ein Element des Arrays enthält eine natürliche Zahl $V(i)$ und einen Zeiger $L(i)$ auf eine Liste. Desweiteren benötigen wir eine Queue Q. Der Algorithmus selbst besteht aus vier Schritten.

1. Initialisierung: Setze alle $V(i) = 0$ und alle $L(i) = NIL$. Dies dauert $O(n)$.

2. Preprocessing: Die Eingabe (Paare) wird durchlaufen. Wird (j,k) gelesen, so wird $V(k)$ um eins inkrementiert, da es ein Element gibt, welches kleiner als k ist und $V(k)$ die Anzahl dieser Elemente zählt.

   k wird in $L(j)$ vorne eingetragen. Hier wird sich praktisch gemerkt, welche Paare mal in der Eingabe standen.

   Dies wird so lange gemacht, bis die Eingabe gelesen wurde. Dies dauert also $O(p)$.

3. Die Queue füllen: Man durchläuft A einmal und schreibt alle i mit $V(i) = 0$ in die Queue Q. Dies dauert $O(n)$.

4. Los gehts: Man entnimmt das erste Element j aus der Queue Q, schreibt es in die Ausgabe und durchläuft die Liste $L(j)$. Für jedes Element k in der Liste $L(j)$ wird $V(k)$ um eins dekrementiert. Wird es dabei Null, so kommt es in die Queue Q. Danach entnimmt man der Queue das nächste Element. Die macht man so lange, bis die Queue leer ist.

   Die Laufzeit ist $O(n+p)$, da man alle n Elemente einmal in die Queue schiebt und Listen der Gesamtlänge p durchläuft.

Ein Beispiel zu diesem Algorithmus findet sich im Wegener-Skript.

- **Welche Möglichkeiten gibt es, um eine spärlich besetzte Matrix mit Hilfe einer linearen Liste zu speichern?**

Wenn man verschiede Matrizen nur addieren möchte, reicht es, jede Zeile der Matrix in einer linearen Liste zu speichern. Dabei stehen jedoch nur die von Null verschiedenen Elemente in der Liste. Und zwar in der Form *(Zeile, Spalte, Wert)*, also $(i,j,A(i,j))$.

Möchte man desweiteren auch noch effizient multiplizieren können, so ist es besser, die Zeilen als auch die Spalten in Listen der obigen Form zu speichern. Eine Zeile, bzw. Spalte sieht also konkret wie folgt aus:

$\rightarrow (2,0,0) \rightarrow (2,3,34) \rightarrow (2,6,-3) \rightarrow NIL$
$\rightarrow (0,3,0) \rightarrow (1,3,1) \rightarrow (2,3,34) \rightarrow NIL$

- **Wie addiert man spärlich besetzte Matrizen, bei denen nur die Zeilen in Listen gespeichert wurden?**

Die Matrizen A und B seien wie oben beschrieben abgespeichert und wir befinden uns in der i-ten Zeile einer jeden Matrix. Seien die Elemente $(i,j,A(i,j)$ und $(i,k,B(i,k))$ erreicht. Wir unterscheiden drei Fälle:

1. $j < k$:

   $(i,j,A(i,j))$ wird in die i-te Zeile der Ergebnismatrix angehängt. Das nächste Element wird in der A-Matrix gesucht.

2. $j > k$:

   $(i,k,B(i,k))$ wird in die i-te Zeile der Ergebnismatrix angehängt. Das nächste Element wird in der B-Matrix gesucht.

3. $j = k$:

   Dies ist der einzige Fall, wo richtig gerechnet wird. Man rechnet $A(i,j)+B(i,k)$ aus und hängt es an die i-te Zeile der Ergebnismatrix an. Danch geht man zum Nachfolger der beiden Elemente.

Wird in einer der beiden Matrizen das Zeilenende (*NIL*) erreicht, so hängt man den Rest der anderen Matrix an die Ergebniszeile. Erreichen beide *NIL* (kann das überhaupt vorkommen?) so wird die Ergebniszeile mit *NIL* abgeschlossen.

Addiert man zwei $n \times m$-Matrizen auf diese Weise miteinander, so beträgt die Laufzeit $O(n+a+b)$, für den Fall, daß die Matrix A insgesamt a von Null verschiedene Einträge hat und die Matrix B insgesamt b.

1.3 Datenstrukturen für Mengen

- **Welche Datenstrukturen gibt es zur Mengendarstellung?**

Die Bit-Vektor-Darstellung und die Listendarstellung.

- **Wann benutzt man die Bit-Vektor-Darstellung und wann die Listendarstellung?**

Die Bit-Vektor-Darstellung kann man benutzen, wenn eine feste Grundmenge $G = \{1, ..., n\}$ gegeben ist und man nur Untermengen dieser Menge darstellen möchte, also $M \subseteq G$.

Die Listendarstellung wird benutzt, wenn die Grundmenge nicht fest bzw. bekannt ist. Oder aber, wenn die Grundmenge sehr groß und die zu darstellenden Menge viel kleiner als die Grundmenge sind. Also $M \subseteq G \wedge G \ll M$.

- **Wie wird die Bit-Vektor-Darstellung implementiert?**

Es gibt zwei Möglichkeiten. Zum einen die Darstellung in einem Array und zum anderen die Speicherung in Maschinenworten.
Sei $|G| = n$. Dann benutzt man ein Array der Länge n und es gilt:

$$A[i] = 1 \Leftrightarrow i \in M, \text{ und } A[i] = 0 \text{ sonst}$$

- **Welche Operationen sind in konstanter Zeit ($O(1)$) durchführbar?**

Man kann in konstanter Zeit ein Objekt der Menge hinzufügen bzw. entnehmen, sowie nachschauen, ob ein sich ein Objekt in der Menge befindet. Im folgenden gelte jeweils $1 \leq i \leq n$:

1. Ist $i \in M$? Antwort: $A[i]$.
2. Füge i zu M hinzu. $A[i] = 1$.
3. Entferne i aus M. $A[i] = 0$.

- **Welche Operationen sind in linearer Zeit bezüglich der Mächtigkeit der Grundmenge ausführbar ($O(n)$ mit $|G| = n$)?**

In linearer Zeit sind die Operationen Vereinigung, Schnitt, Differenz und symetrische Differenz ausführbar. Sei das Array A_i jeweils die Darstellung der Menge M_i. Auch hier gelte $1 \leq i \leq n$:

1. $M = M_1 \cup M_2$. $A[i] := A_1[i] \vee A_2[i]$.
2. $M = M_1 \cap M_2$. $A[i] := A_1[i] \wedge A_2[i]$.
3. $M = M_1 - M_2$. $A[i] := A_1[i] \wedge (\neg A_2[i])$.
4. $M = M_1 \Delta M_2$. $A[i] := A_1[i] \oplus A_2[i]$. (Symmetrische Differenz)

- **Wie werden diese Operationen bei der Speicherung in Maschinenwör-tern implementiert? Und wie effizient sind diese?**

Angenommen in ein Maschinenwort passen w Bits. Wenn auf jedes Bit eines Wortes in konstanter Zeit zugegriffen werden kann, so sind auch die ersten drei Operationen (Einfügen, Entfernen und Nachschauen) in konstanter Zeit durchführbar. $A[i]$ steht dann im $\lceil \frac{i}{w} \rceil$-ten Wort an der Position $i \bmod w$, wenn die Positionen mit $0, ..., w-1$ durchnummeriert sind. Sonst logischerweise an Position $(i \bmod w) + 1$.

Im allgemeinen sind auch die Booleschen Operationen $\wedge, \vee, \neg, \oplus$ auf ganzen Wörtern in konstanter Zeit durchführbar. Die letzten Operationen (Vereinigung, Schnitt, Differenz, Symmetrische Differenz) zerfallen dann in jeweils in $\lceil \frac{n}{w} \rceil$ Operationen.

- **Welche Zeit brauchen diese Operationen bei der Listendarstellung, wenn man davon ausgeht, daß die Objekte zu einer nicht geordneten Menge gehören?**

Die ersten drei Operationen, die jeweils nur ein Objekt betreffen, brauchen Zeit *O(LENGTH(L))*. Bei den letzten vier Operationen muß jedes Element aus L_1 in L_2 gesucht werden und umgekehrt. Die Rechenzeit beläuft sich in diesem Fall also auf $O(LENGTH(L_1) \cdot LENGTH(L_2))$.

Bei der Vereinigung gibt es noch den Ausweg, $UNION(L_1, L_2, L)$ zu benutzen. Dabei kommt es jedoch zu Mehrfachdarstellung von Elementen, wenn M_1 und M_2 nicht disjunkt sind.

- **Wie sieht es aus, wenn die Objekte aus einer geordneten Menge stammen?**

Hier brauchen Operationen, welche ein Objekt betreffen weiterhin *O(LENGTH(L))* Zeit. Jedoch kann man jetzt auch nach erfolglosen Suchen abbrechen, sobald ein Objekt gefunden wurde, welches größer als das gesuchte Objekt ist.

Bei den anderen Operationen kann man das Reißverschlußverfahren benutzen, welches auch für die Zeilenaddition spärlich besetzter Matrizen benutzt wurde. Dann kosten diese Operationen also Zeit $O(LENGTH(L_1) + LENGTH(L_2))$.

1.4 Datenstrukturen für Bäume

- **Gebe Beispiele für Bäume in der Informatik!**

Bäume kommen sehr häufig in der Informatik vor:

1. Syntaxbäume
2. Suchbäume
3. Arithmetische Ausdrücke
4. Branch-and-Bound-Bäume

- **Definiere einen gewurzelten Baum mit Kanten, die von der Wurzel weg gerichtet sind!**

Ein solcher Baum besteht aus:

1. Endlicher Knotenmenge V, häufig $V = \{1, ..., n\}$
2. Kantenmenge $E \subseteq V \times V - \{(i,i) | i \in V\}$
3. Einem Knoten r, genannt Wurzel, für den es kein $(\cdot, r) \in E$ gibt.
4. Jeder Knoten v hat $ingrad(v) = 1$, außer die Wurzel.

- **Wie viele Kanten haben Bäume?**

$|V| - 1$, da es für alle Knoten $v \neq r$ genau einen Knoten w mit $(w,v) \in E$ gibt.

- **Was ist ein Elter und was ist ein Kind?**

Gilt $(w,v) \in E$, so ist w Elter von v. v heisst dann Kind von w.

- **Was sind Geschwister?**

Gilt $v \neq v'$ und $(w,v) \in E$, sowie $(w,v') \in E$, so sind v und v' Geschwister.

- **Wann heißt ein Knoten v Nachfolger von v_0? Beziehungsweise wann heißt v_0 Vorgänger von v? Was ist in diesem Zusammenhang ein Weg (oder Pfad)?**

Wenn es $v_1, ..., v_m = v$ gibt, so daß $(v_0, v_1), (v_1, v_2), ..., (v_{m-1}, v_m) \in E$ sind. Diese Kante beschreiben dann einen Weg (Pfad) der Länge m von v_0 nach v_m.

- **Was ist der *ingrad(v)* bzw. *outgrad(v)* eines Knotens v?**

Der *ingrad(v)* ist die Zahl der Eltern, die ein Knoten v hat, also die Anzahl $(\cdot, v) \in E$. Der *outgrad(v)* ist die Zahl der Kinder von v und somit die Anzahl $(v, \cdot) \in E$.

- **Wie verhält es sich bei Bäumen mit dem *ingrad(v)* und dem *outgrad(v)*?**

Bei Bäumen gilt *ingrad(v)*=1 für $v \neq r$, sowie *ingrad(r)*=0. Der *outgrad(v)* eines Knotens ist gleich der Anzahl seiner Kinder, d.h. für Blätter gilt: *outgrad=0.*

- **Was sind Blätter?**

Ein Knoten v heißt Blatt, wenn gilt *outgrad(v)*=0, wenn der Knoten also keine Kinder hat.

- **Was ist die Tiefe eines Knotens v in einem Baum T?**

Die Tiefe eines Knotens v in einem Baum T ist die Länge des eindeutiges Weges von der Wurzel r zu v.

- **Was ist die Tiefe eines Baumes?**

Die Tiefe eines Baumes ist die maximale Tiefe eines Knotens, also die maximale Tiefe eines Blattes.

- **Was ist der Unterschied zwischen *outgrad-k* beschränkt und k-är?**

Ein Baum heißt *outgrad-k* beschränkt, wenn kein Knoten mehr als *k* Kinder hat. Ein Knoten kann also weniger ausgehende Kanten haben.

Ein Baum heißt *k*-är, wenn alle inneren Knoten, das sind die Knoten, die keine Blätter sind, genau *k* Kinder haben. Ein unärer Baum (1-är) ist demnach eine Lineare Liste. Ein binärer Baum ist ein 2-ärer Baum. Meist bezeichnet man mit binären Bäumen *outgrad*-2 beschränkte Bäume, was sehr schlampig ist.

- **Wie viele Knoten hat ein vollständiger *k*-ärer Baum der Tiefe *d*?**

Da alle Blätter die Tiefe *d* haben, gilt für die Anzahl Anz(T) der Knoten:

$$\sum_{n=0}^{d} k^n = 1 + k + ... + k^d = \frac{(k^{d+1} - 1)}{(k-1)}$$

Häufig ordnet man die Kinder eines Knotens, so daß man für $1 \leq i \leq$ *outgrad(v)* vom *i*-ten Kind sprechen kann. Jeder Knoten *v* definiert den Teilbaum, der aus *v* und seinen Nachfolgern besteht.

Sei $k =$ *outgrad(v)* und $v_1, ..., v_k$ die Kinder von *v*, dann zerfällt $T(v)$ in die Wurzel *v* und die Teilbäume $T(v_1), ..., T(v_k)$.

- **Wie sind Postorder, Preorder, Inorder und Levelorder definiert?**

Diese vier Ordnungen sind rekursiv definiert. $T = (V, E)$ sei ein geordneter Baum mit Wurzel $r(T)$ und den Kindern der Wurzel $v_1, ..., v_k$, wobei $k \geq 0$.

1. $\text{post}(T) := \text{post}(T(v_1))...\text{post}(T(v_k))r(T)$
2. $\text{pre}(T) := r(T)\text{pre}(T(v_1))...\text{pre}(T(v_k))$
3. $\text{in}(T) := \text{in}(T(v_1))r(T)\text{in}(T(v_2))...\text{in}(T(v_k))$
4. $\text{lev}(T) := r(T)\text{lev}(T(v_1))r(T)\text{lev}(T(v_2))...r(T)\text{lev}(T(v_k))r(T)$

- **Läßt sich aus einer dieser Ordnungen ein Baum T wieder eindeutig rekonstruieren?**

Aus Postorder, Preorder und Inorder allein läßt sich ein Baum nicht rekonstruieren. Es gibt *n*! verschiedene Reihenfolgen der Ausgabe. Damit lassen sich jedoch nur alle linearen Listen beschreiben. Aus der Levelorder eines Baumes kann man diesen jedoch rekonstruieren.

- **Welche Länge haben Postorder, Preorder und Inorder für $V = \{1, ..., n\}$? Welche Länge für Levelorder?**

Für Postorder, Preorder und Inorder hat die Ausgabe die Länge *n*. In der Levelorder tritt jeder Knoten *v* $(outgrad(v) + 1)$-mal auf. Also hat eine Levelorder von T die Länge $2n - 1$.

- **Beweise, daß man aus der Levelorder eines Baumes T diesen wieder eindeutig rekonstruieren kann!**

Dies wird per Induktion bewiesen: Besteht der Baum nur aus einem Knoten, so wird nur dieser ausgegeben. Ansonsten stehen zwischen den „Wurzelausgaben" die Levelorder der Teilbäume, welche nach Induktionsvorraussetzung einen Baum eindeutig beschreiben.

- **Kann man mit Preorder und Inorder einen beliebigen Baum beschreiben?**

Nein. Ein Gegenbeispiel findet sich im Skript auf Seite 35. Jedoch kann man mit Preorder und Inorder einen **binären** Baum beschreiben.

Beweis:

$$\begin{aligned} pre(T) &= r(T)\,pre(T_1)\,pre(T_2) \\ in(T) &= in(T_1)\,r(T)\,in(T_2) \end{aligned}$$

Anhand der Preorder-Ausgabe stellt man fest, welcher Knoten die Wurzel $r(T)$ ist. Diese sucht man in der Inorder-Ausgabe. Jetzt kann man $in(T_1)$ und $in(T_2)$ trennen und kennt die Knoten, die in beiden Teilbäumen sind. Daher kann man auch in der Preorder-Ausgabe $pre(T_1)$ und $pre(T_2)$ diffenzieren. Dieses Verfahren wird nun rekursiv fortgesetzt, bis man auf Bäume mit einem Knoten stößt.

Aufgrund der **Symmetrie** von Pre- und Postorder gilt dasselbe für Postorder und Inorder.

- **Mit Hilfe welcher beiden Order kann man einen beliebigen Baum repräsentieren?**

Mit Preorder und Postorder. **Beweis**:

Gegeben seien die Definitionen von Pre- und Postorder. Die Wurzel kann man anhand dieser Darstellung sofort identifizieren, sie steht in der Preorder ganz vorne und in der Postorder ganz hinten. Der erste Knoten in $pre(T_1)$ ist die Wurzel von T_1. Damit kann man $post(T_1)$ abtrennen. Da $post(T_1)$ jetzt bekannt ist und damit auch die Knoten, die dieser Unterbaum beinhaltet, kann man auch $pre(T_1)$ abtrennen. Dieses Verfahren setzt man fort, bis man alle Unterbäume abgetrennt hat und benutzt dieses Verfahren rekursiv für die Unterbäume, bis man nur noch auf einknotige Bäume stößt.

Beispiel:

Gegeben sei der Baum T_1 aus dem Wegener-Skript auf Seite 35. Dann gilt:

$$\begin{aligned} pre(T_1) &= 1,2,3,4,5,6 \\ post(T_1) &= 2,4,5,3,6,1 \end{aligned}$$

Die Wurzel ist offensichtlich 1. Das erste Kind der Wurzel ist 2. In der Postorder steht 2 ganz links, also hat dieser Knoten keine Kinder. Also gehen wir zum zweiten Kind 3. Zwischen der abgetrennten 2 und der 3 stehen noch die Knoten 4 und 5. Die 2 hat also noch Unterbäume, die man rekursiv behandeln kann. Wir überspringen 4 und 5 in der Preorder und finden nur noch die 6. Also hat der Knoten 1 drei Kinder. Dies sind 2, 3 und 6, wobei die drei selbst noch zwei Kinder hat: 4 und 5.

- **Welche Operationen für Bäume werden häufig unterstützt?**

1. *PARENT(x,T):* Berechne den Elter zu x im Baum T. Wenn x die Wurzel ist lautet die Antwort *nil*.
2. *CHILD(x,i,T):* Berechne das i-te Kind von x in T. Wenn x weniger als i Kinder hat lautet auch hier die Antwort *nil*.
3. *LCHILD(x,T):* Berechne das erste Kind von x in T. Also *CHILD(x,1,t)*.
4. *RCHILD(x,T):* Berechne das letzte Kind von x in T.
5. *ROOT(T):* Berechne die Wurzel von T.

6. *CONCATENATE(x,$T_1, ..., T_m$):* Erzeuge einen Baum mit Wurzel x und den m Unterbäumen $T_1, ..., T_m$. Oder wie im Skript. Mit m Kindern $v_1, ..., v_m$, so daß $T(v_1) = T_1, ..., T(v_m) = T_m$ ist.

7. *DEPTH(x,T)*: Berechne die Tiefe von x in T.

8. *SIZE(T):* Berechne die Zahl der Knoten in T.

• Welche Datenstrukturen für Bäume gibt es? Welche Operationen unterstützen sie besonders gut, welche nicht? Wie gut ist die Speicherplatzausnutzung?

1. Die Knoten werden in einem Array abgespeichert. Für jeden Knoten wird sein Elter angegeben. Dies kann mit Hilfe von Zeiger- oder Arraydarstellung passieren. Zusätzlich kann man *root(T)* und *size(T)* abspeichern.

   a) *PARENT:* $O(1)$.

   b) *CHILD:* $O(n)$, da das Array durchlaufen werden muß, um zu schauen, welche Knoten x als *PARENT* angegeben haben.

   c) *ROOT:* $O(1)$, wenn abgespeichert, sonst $O(DEPTH(T))$, da man von einem Knoten den *PARENT*-Zeigern folgen muß, bis man zu *nil* gelangt.

   d) *CONCATENATE:* In $O(m)$, wenn die Wurzel gefunden wurde. Die einzelnen Arrays werden hintereinander gehängt und x als Elter der Wurzeln angegeben. Hier empfiehlt sich die Zeiger-Variante, da man sonst die Elterangaben in den anzuhängenden Arrays korrigieren muß.

   e) *DEPTH:* In $O(DEPTH(T))$, da dies ähnlich ist, wie das Finden der Wurzel, wenn diese nicht abgespeichert ist. Das Finden der Wurzel terminiert den Algorithmus praktisch.

   f) *SIZE:* In $O(1)$, wenn abgespeichert. Sonst anscheinend die Länge des Arrays. Also evtl. $O(n)$?

   Hierbei wird kein Speicherplatz verschwendet. Es werden die Knoten selbst abgespeichert und durch die Angabe des Elters die Kanten.

2. Wieder wird ein Array verwendet. Diesmal speichert man jedoch in einer geordneten Liste die Kinder. Hier drehen sich die Probleme praktisch um:

   a) *PARENT:* $O(n)$, da man alle Listen durchlaufen muß, um zu schauen, ob der Knoten Elter von x ist.

   b) *CHILD:* $O(i)$(???), da man bis zum i-ten Element in der Liste laufen muß, um das Kind festzustellen.

   c) *ROOT:* $O(1)$, wenn abgespeichert, sonst $O(n)$.

   d) *CONCATENATE:* $O(m)$. Die Kinderliste des Wurzelknotens wird neu aufgebaut, indem die Wurzeln $r_1, ..., r_m$ aufgenommen werden.

   e) *DEPTH:* Ist jetzt auf jeden Fall komplizierter.

   f) *SIZE:* In $O(1)$, wenn abgespeichert. Sonst anscheinend die Länge des Arrays. Also evtl. $O(n)$?

Hier wird für die Zeiger zusätzlicher Platz verbraucht. Da es bei n Knoten $n-1$ Zeiger gibt, wird $O(n-1) = O(n)$ zusätzlicher Platz gebraucht.

3. Kombination aus (1) und (2). Die Vorteile werden auf Kosten des Speicherplatzes kombiniert. Auch hier zusätzlich gebrauchter Platz von $O(n)$.

4. Ist der Baum *outgrad-k*-beschränkt, so ist also die größtmögliche Zahl an Kindern bekannt. Dann kann man auf die Listen aus (2) und (3) verzichten und Arrays benutzen, die ja einen direkten Zugriff in $O(1)$ ermöglichen.

   Ist der Baum outgrad(k)-beschränkt, so benutzt man für jeden Knoten ein Array der Länge $(k+1)$. Dabei stehen an den ersten k Positionen, die eventuell vorhandenen Kinder und an der letzten Position $(k+1)$ der Elter des Knoten. Hat ein Knoten nicht alle k Kinder, so sind einige Einträge vor $(k+1)$ *NIL*. Welche Operationen werden in welcher Zeit unterstützt?

   a) *PARENT:* Ist in $O(1)$ berechenbar.

   b) *CHILD:* Ist in $O(1)$ berechenbar.

   c) *LCHILD:* Ist, da das Kind an der ersten Arrayposition gesucht wird, in $O(1)$ berechenbar.

   d) *ROOT:* Ist, wenn die Wurzel explizit abgespeichert wurde, in $O(1)$ berechenbar.

   e) *SIZE:* Ist, wenn explizit für jeden Baum abgespeichert, in $O(1)$ berechenbar.

   f) *CONCATENATE:* Ist in $O(m) = O(k)$ berechenbar, da nur ein neuer Knoten erzeugt werden muß, der die Wurzeln, der zu konkatenierenden Bäume, als Kinder hat.

   g) *DEPTH:* Ist in $O(DEPTH(T))$ berechenbar.

   h) *RCHILD:* Ist in $O(\log_2 k)$ berechenbar, wenn der rechteste Knoten nicht explizit abgespeichert ist. In diesem Fall muß man den Knoten nämlich mit binärer Suche bestimmen.

   Wie gut ist die Speicherplatzausnutzung? Da jede Kante einmal als *PARENT*-Kante und einmal als *CHILD*-Kante vertreten ist, und es $(n-1)$ Kanten gibt, sind also $2(n-1)$ Kanten von *NIL* verschieden. Insgesamt ist Platz für $n(k+1)$ Kanten, da es n Knoten gibt, die jeweils $(k+1)$ Arrayplätze haben. Wieviele Zeiger sind also *NIL*? Logischerweise die Differenz: $n(k+1) - 2(n-1) = nk + n - 2n + 2 = kn - n + 2$. In welchem Verhältnis stehen die *NIL*-Zeiger zu den

anderen Zeigern?

$$\begin{aligned}
\frac{nk-n+2}{n(k+1)} &= \frac{nk+(n-n)-n+2}{n(k+1)} \\
&= \frac{n(k+1)-2n+2}{n(k+1)} \\
&= \frac{n(k+1)}{n(k+1)} - \frac{2n-2}{n(k+1)} \\
&= 1 - \frac{2n}{n(k+1)} + \frac{2}{n(k+1)} \\
&= 1 - \frac{2}{k+1} + \frac{2}{n(k+1)} \\
&> 1 - \frac{2}{k+1}
\end{aligned}$$

Die letzte Abschätzung gilt unter anderem wegen $\lim_{n\to\infty} \frac{2}{n(k+1)} = 0$. Je größer k gewählt wird, desto größer also der Anteil der *NIL*-Zeiger. Für $k = 1$ gibt es kaum *NIL*-Zeiger, dann ist der ,,Baum" jedoch eine lineare Liste. Für $k = 2$ gilt $1 - \frac{2}{2+1} = 1 - \frac{2}{3} = \frac{1}{3}$. Ca. 33% der Zeiger sind also *NIL*-Zeiger. Mit dieser Zahl kann man noch leben und außerdem hat man es mit richtigen Bäumen zu tun. Also versuchen wir in (5) beliebige Bäume durch binäre Bäume zu repräsentieren.

5. Sei T ein beliebiger Baum. Ein Knoten v in diesem Baum hat ein Array mit drei Elementen. Diese sind:

   a) Zeiger auf den *PARENT*-Knoten von v. In der Wurzel *NIL*.

   b) Zeiger auf das linkeste Kind von v. In Blättern *NIL*.

   c) Zeiger auf das in der linearen Ordnung auf v folgende Geschwist. Auch eventuell *NIL*.

   Welche Zeit brauchen die Operationen?

   a) *PARENT:* Ist in $O(1)$ berechenbar.

   b) *LCHILD:* Ist in $O(1)$ berechenbar.

   c) *CHILD(· ,i,T):* Ist in Zeit $O(i)$ berechenbar. Man geht von Knoten, dessen i-tes Kind man berechnen möchte, zu seinem linkesten Kind und von dort aus $(i-1)$ mal zum Knoten, der an der dritten Arrayposition gespeichert ist.

   d) *RCHILD(x,T):* Braucht, wie man in der Erklärung zu *CHILD* sieht, Zeit *O(outgrad(x)).*

- **Gebe eine iterativen Algorithmus zur Inorder-Ausgabe an!**

Der Algorithmus läßt sich wie folgt beschreiben:
Ein Stack sei initialisiert und der aktuelle Knoten sei die Wurzel. Lege den aktuellen Knoten auf den Stack. Wenn der aktuelle Knoten ein linkes Kind hat, setze dieses Kind als aktuellen Knoten und lege ihn auf den Stack. Hat ein Knoten kein linkes

Kind, so gebe ihn aus und nimm ihn vom Stack. Hat der aktuelle Knoten ein rechtes Kind, so mache es zum aktuellen Knoten und lege es auf den Stack. Hat der Knoten kein rechtes Kind, so betrachte das nächste Element auf dem Stack. Dies macht man so lange bis der Stack leer ist.

Man simuliert mit dem Stack natürlich auch nur den Rekursionsstack, der aufgebaut wird, wenn das Problem rekursiv gelöst wird.

- **Was ist ein Inorder-gefädelter binärer Baum?**

Bei Inorder-gefädelten binäre Bäumen nutzt man die Zeiger besser aus. Man unterscheidet bei ihnen zwischen echten Zeiger und unechten Zeigern. Echte Zeiger zeigen auf die Kinder eines Knotens, während unechte linke Zeiger auf den Inorder-Vorgänger zeigen und unechte linke Zeiger auf den Inorder-Nachfolger. Gibt es keinen Vorgänger bzw. Nachfolger, so zeigen diese Zeiger auf *NIL*. Ob ein Zeiger echt oder unecht ist, wird durch boolesche Variablen verwaltet, die nicht viel Platz beanspruchen.

- **Gebe einen Algorithmus zur Inorder-Ausgabe eines Inorder-gefädelten binären Baumes an!**

1. Der aktuelle Knoten sei zu Beginn die Wurzel. Gehe zu (2).
2. Falls der linke Zeiger echt ist, mache das Kind am linken Zeiger zum aktuellen Knoten und gehe zu (2).

   Sonst gebe das Datum am aktuellen Knoten aus und gehe zu (3).
3. Ist der rechte Zeiger echt, so mache das Kind des rechten Zeigers zum aktuellen Knoten und gehe zu (2).

   Ist der rechte Zeiger unecht, so gehe zu (4).
4. Ist der rechte nicht *NIL*, so schreibe das Kind des unechten rechten Zeigers in die Ausgabe und mache es zum aktuellen Knoten. Dann gehe zu (3).

   Ist der rechte unechte Zeiger *NIL*, ist der Algorithmus beendet.

Die Laufzeit des Algorithmus ist $O(n)$, da jeder Knoten höchstens zweimal besucht wird. Nämlich einmal beim Herunterhangeln und einmal beim Heraufhangeln. Der Algorithmus arbeitet auch korrekt, da ein Datum erst ausgegeben wird, wenn der komplette linke Teilbaum abgearbeitet ist.

- **Gebe einen Algorithmus zur Preorder-Ausgabe eines Inorder-gefädel-ten binären Baumes an!**

Zu Beginn sei die Wurzel der aktuelle Knoten.

1. Gebe den aktuellen Knoten aus und gehe zu (2).
2. Ist der linke Zeiger echt, so mache ihn zum aktuellen Knoten und gehe zu (1).

   Ist der linke Zeiger unecht, so gehe zu (3).
3. Ist der rechte Zeiger des Knotens echt, so gebe das Kind des rechten Zeigers aus und mache es zum aktuellen Knoten. Gehe zu (2).

   Ist der rechte Zeiger unecht, gehe zu (4).

4. Verfolge die rechten Zeiger so lange zurück, bis zum ersten Mal ein echter Zeiger gefunden wird. Gebe das Kind des ersten echten rechten Zeigers aus und mache es zum aktuellen Knoten.

   Ist ein unechter rechter Zeiger *NIL*, ist der Algorithmus beendet.

Auch hier ist die Laufzeit $O(n)$, da jeder Knoten höchstens zweimal besucht wird.

1.5 Datenstrukturen für Graphen

- **Wie ist ein Graph definiert?**

Ein Graph $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer endlichen Kantenmenge E, mit $E \subseteq V \times V$.

- **Was ist der Unterschied zwischen einem gerichteten und einem ungerichteten Graphen?**

Bei einem gerichteten Graphen bedeutet $e = (v, w) \in E$, daß eine Kante vom Knoten v nach w verläuft nicht jedoch zwingend umgekehrt. Bei ungerichteten Graphen bedeutet $e = (v, w) \in E$, daß eine Kante zwischen beiden Knoten verläuft.

- **Was hat es mit Adjazenz und Inzidenz auf sich?**

Zwei Knoten heißen adjazent, wenn zwischen ihnen eine Kante verläuft. Ein Knoten ist mit einer Kante inzident, wenn er auf der Kante liegt. Z.B. ist der Knoten v inzident mit der Kante (v, w), wenn in einem Graphen eine Kante zwischen v und w besteht.

- **Was ist der grad(v), ingrad(v) und outgrad(v) eines Knotens v?**

Im ungerichteten Graphen ist der $grad(v)$ eines Knotens v gleich der Anzahl der Kanten, mit denen er inzident ist. In einem gerichteten Graphen ist der $ingrad(v)$ gleich der Anzahl der Kanten $(\cdot, v) \in E$, bzw der $outgrad(v)$ gleich die Zahl der Kanten $(v, \cdot) \in E$.

- **Was ist ein Nachfolger bzw. Vorgänger eines Knotens v?**

Die Nachfolger eines Knotens v sind die Knoten w, für die $(v, w) \in E$ gilt. v ist dann der Vorgänger aller dieser Knoten.

- **Was ist ein Weg?**

Ein Weg in einem Graphen ist eine Folge $v_0, ..., v_k$, falls gilt:

$$(v_{i-1}, v_i) \in E, 1 \leq i \leq k$$

- **Wann heißt ein Weg einfach?**

Ein Weg heißt einfach, wenn höchstens Anfangs- und Endknoten gleich sind, d.h. daß kein Knoten zweimal besucht wird.

- **Was ist ein Kreis?**

Ein Kreis ist ein einfacher Weg, bei dem Anfangs- und Endpunkt übereinstimmen. Hat ein Graph G keinen Kreis, so heißt er azyklisch. In ungerichteten Graphen werden Kreise der Länge 2 jedoch ausgeschlossen.

- **Was sind Zusammenhangskomponenten bzw. starke Zusammenhangskomponenten?**

Durch folgende Äquivalenzrelation werden in einem ungerichteten Graphen Zusammenhangskomponenten definiert: Zwei Knoten v und w heißen zusammenhängend:

$$v \approx w : \exists \text{ ein Weg zwischen } v \text{ und } w$$

Für gerichtete Graphen werden durch folgende Äquivalenzrelation starke Zusammenhangskomponenten definiert:

$$v \approx w : \exists \text{ ein Weg zwischen } v \text{ und } w \text{ und } \exists \text{ ein Weg zwischen } w \text{ und } v$$

- **Zeige, daß diese Relationen Äquivalenzrelationen sind!**

Zu zeigen sind Reflexivität, Symmetrie und Transitivität:

1. $\forall v \in V : v \approx v$ (Reflexivität)

   Es existiert (trivialerweise) ein Weg von einem Knoten v zu sich selbst.

2. $\forall v, w \in V : v \approx w \Rightarrow w \approx v$ (Symmetrie)

   Gegeben durch die Definitionen.

3. $\forall v, w, x \in V : v \approx w \wedge w \approx x \Rightarrow v \approx x$ (Transitivität)

   Wenn es einen Weg von v zu w gibt und einen Weg von w zu x, so gibt es auch einen Weg von v zu x.

Die Knotenmenge zerfällt also in Äquivalenzklassen. In ungerichteten Graphen sind dies die Zusammenhangskomponenten und in gerichteten Graphen die starken Zusammenhangskomponenten.

• **Welche Datenstrukturen gibt es zur Speicherung von Graphen und wie funktionieren diese?**

In folgenden sei $|V| = n$ und $|E| = m$ und die Knoten und Kanten entsprechend durchnumeriert.

1. Inzidenzmatrix:

   Es handelt sich bei dieser Art der Speicherung um eine $n \times m$-Matrix. Die Einträge sind für ungerichtete Graphen 0 oder 1 und für gerichtete Graphen 0,1 oder 2.

   Der Eintrag $I(i, j) = 1$, wenn der Knoten i mit der Kante j inzident ist, sonst ist der Eintrag 0.

   Bei gerichteten Graphen ist der Eintrag $I(i, j) = 1$, falls i der Anfangsknoten ist. Ist i der Endknoten, so ist der Eintrag $I(i, j) = 2$. Wie bei ungerichteten Graphen 0 sonst.

   Da eine Kante nur zwischen zwei Knoten verlaufen kann, ist die Matrix also spärlich besetzt. Zudem braucht diese Art der Speicherung viel Platz.

2. Adjazenzmatrix:

   Es handelt sich um eine $n \times n$-Matrix. Es gilt $A(i, j) = 1$, wenn in einem gerichteten Graphen eine Kante zwischen den Knoten i und j verläuft. Sind die Kanten bewertet, so kann statt einer 1 gleich der Wert gespeichert werden. Dabei gilt $A(i, j) = \infty$, wenn keine Kante zwischen zwei Knoten i und j verläuft.

   Bei ungerichteten Graphen braucht man aufgrund der Symmetrie nur die Werte unterhalb der Hauptdiagonalen speichern. Auf jeden Fall beträgt der Speicherbedarf $\Theta(n^2)$.

   Die Frage, ob es eine Kante zwischen zwei Knoten gibt kann in $O(1)$ beantwortet werden.

   Der Grad eines Knotens i kann in $O(n)$ berechnet werden, da nur einmal die i-te Zeile und einmal die i-te Spalte durchlaufen werden muß. Also bei n Knoten der Aufwand $2n = O(n)$.

   Algorithmen, die alle Kanten betrachten, haben jedoch eine Laufzeit von $\Omega(n^2)$ (mindestens n^2).

3. Adjazenzlisten:

   Im Grunde genommen, sind Adjazenzlisten die komprimierte Darstellung spärlich besetzter Adjazenzmatrizen, d.h. wenn $|E| = o(n^2)$.

   Wir haben ein Array der Länge $n = |V|$. Für jeden Knoten verwalten wir eine Liste, in der seine Nachfolger gespeichert sind. Der Speicherplatzbedarf beträgt $O(n+m) = O(|V| + |E|)$.

   Ob eine Kante $(i, j) \in E$ existiert, kann jedoch nicht mehr in $O(1)$ entschieden werden, da dazu die Liste des Knotens i durchlaufen werden muß. Sei l_i die Länge der Liste des i-ten Knotens. Dann braucht man Zeit $O(l_i)$, um zu entscheiden, ob eine Kante zwischen den Knoten $i, j \in V$ verläuft. Die selbe Zeit braucht man zur Bestimmung des Outgrads. Zur Bestimmung des Grades eines Knotens braucht man Zeit $\Theta(m) = \Theta(|E|)$, da man sich alle Knoten anschauen muß. Daraus folgt auch, daß die Bestimmung des ingrades genausolange dauert. Es sei denn, es sind auch noch Adjazenzlistem für eingehende Kanten gespeichert. Dann würde auch die Berechnung des Grades in Zeit $O(l_i)$ gehen.

- **Wie arbeitet der DFS-Algorithmus für ungerichtete Graphen?**

Bei der Tiefesuche (DFS=**D**epth-**F**irst-**S**earch) wird so weit wie möglich in die Tiefe gesucht, bevor ein Backtracking erfolgt. In ungerichteten Graphen werden die Kanten dabei in Tree- und Back-Kanten klassifiziert.

Jeder Knoten, der durch einen Aufruf erreicht wird, bekommt eine Nummer zugeordnet, sofern er noch keine hat. Können keine Knoten mehr erreicht werden, wird die Suche von einem noch nicht besuchten Knoten (hat noch keine Nummer) neu gestartet. Die Suche wird logischerweise so oft neu gestartet, wie es Zusammenhangskomponenten gibt. Die Tree-Kanten (kurz T-Kanten) stellen dabei die Zusammenhangskomponenten dar. Wenn es mehrere Zusammenhangskomponenten gibt, bilden die T-Kanten also einen Wald.

Wie sieht jetzt ein konkreter Algorithmus für DFS aus? Der Graph $G = (V,E)$ sei gegeben durch ein Knotenarray mit Adjazenzlisten. Das Rahmenprogramm sieht wie folgt aus:

1. Initialisierung:

   Eine globale Variable $i = 0$ wird initialisiert. Desweiteren gelte $T = \emptyset$ und $B = \emptyset$. Die Nummern aller Knoten $v \in V$ wird $num(v) = 0$ gesetzt.

2. Suche Zusammenhangskomponenten:

   Für alle $x \in V$ wird nun geprüft, ob $num(v) = 0$ gilt, dieser Knoten also noch nicht besucht wurde. Für den ersten Knoten im Graphen, insofern G nicht leer ist, gilt dies auf jeden Fall. Für alle Knoten, für die $num(v) = 0$ gilt, wird DFS mit $DFS(x, 0)$ aufgerufen.

Allgemein bedeute $DFS(x,v)$, daß der Knoten x von v aus erreicht wurde. $DFS(x,0)$ bedeute, daß x der erste Knoten einer Zusammenhangskomponente sei.

$DFS(v,u)$ sei aufgerufen worden:

1. Nummer zuordnen:

   $i = i + 1$, $num(v) = i$

   Der Knoten der erreicht wurde bekommt eine Nummer ungleich 0 zugeordnet.

2. Adjazenzliste betrachten:

   $\forall w \in V$:

   Wenn $num(w) = 0$ ist, w also noch nicht betrachtet wurde, füge die Kante (v,w) zu den T-Kanten hinzu ($T = T \cup \{(v,w)\}$) und starte $DFS(w,v)$. Sonst wenn $num(w) < num(v)$ gilt und $w \neq u$, füge die Kante (v,w) zu den B-Kanten hinzu. Denn dann ist man nicht den Weg u,v,u gegangen.

- **Welche Laufzeit hat DFS?**

Die Laufzeit von DFS beträgt $O(n+m) = O(|V| + |E|)$. Denn alle Knoten werden genau einmal betrachtet ($num(v) = 0$?) und von jedem Knoten wird die Adjazenzliste genau einmal durchlaufen.

• **Zeige, daß jede Kante in einem ungerichteten Graphen nur eine Klassifikation verursacht.**
Zuerst sei einmal gesagt, daß diese Forderung mehr als sinnvoll ist, da sonst mindestens die Hälfte aller Kanten auch Back-Kanten wären, bzw. alle Knoten die miteinander verbunden wären, hätten eine B-Kante. Dies wird dadurch verhindert, daß DFS jeweils in der Form $DFS(v,u)$ aufgerufen wird. Man weiß also von welchem Knoten man gekommen ist. Dieser Knoten befindet sich auch auf jeden Fall in der Adjazenzliste von v, d.h. $u \in Adj(v)$. Es gilt also $num(w) < num(v)$, jedoch mit $w = u$. Also wird für diese Kanten garnichts gemacht.

Die anderen Fälle klassifizieren die restlichen Kanten auf jeden Fall entweder als Tree- oder Back-Kante.

• **Wie kann man einen Graphen darauf testen, ob er kreisfrei ist.**
Indem man einen DFS macht und schaut, ob $B = \emptyset$ gilt. Back-Kanten können nicht zwischen den Bäumen des T-Waldes verlaufen, da es sonst kein Wald wäre. Also schließt eine B-Kante einen Kreis in einem Baum.

• **Wie funktioniert DFS für gerichtete Graphen?**
Für gerichtete Graphen verläuft der DFS ähnlich, wie für ungerichtete Graphen. Da jetzt jedoch nicht mehr trivialerweise für zwei Knoten v,w gilt: $(v,w) \in E \Rightarrow w \in Adj(v) \wedge v \in Adj(w)$, wird nun jede Kante einzeln klassifiziert.

Dazu brauchen wir jedoch für jeden Knoten $v \in V$ noch einen Hilfsparameter $\alpha(v)$. Dieser wird zu Beginn für jeden Knoten mit $\alpha(v) = 0$ initialisiert. Während des Aufrufs $DFS(v,\cdot)$ ist $\alpha(v) = 1$. Ist $DFS(v,\cdot)$ abgearbeitet, gilt $\alpha(v) = 2$. Wir werden die Kantenmenge jetzt in vier disjunkte Kantenmengen einteilen: Den Tree-, Forward-, Back-, und Cross-Kanten.

Es werde die Kante (v,w) betrachtet:

1. Tree-Kanten: Es gilt $num(w) = 0$. Dann wird (v,w) Tree-Kante, da ein neuer Knoten erreicht wurde. Danach wird $DFS(w,\cdot)$ aufgerufen.

2. Forward-Kanten: Es gilt $num(w) \neq 0$, mit $num(w) > num(v)$. Dann wird (v,w) Forward-Kante, denn w wurde schon komplett bearbeitet, kann jedoch von einem Knoten, der noch nicht abgearbeitet wurde, erreicht werden.

3. Back-Kanten: Es gilt $num(w) \neq 0$, mit $num(w) < num(v)$ und $\alpha(w) = 1$. Dann wird (v,w) Back-Kante, denn es wurde ein Knoten erreicht, der noch nicht abgearbeitet wurde, dessen DFS-Aufruf jedoch früher gestartet wurde. Also verläuft diese Kante im T-Baum rückwärts.

4. Cross-Kanten: Es gilt $num(w) \neq 0$, mit $num(w) < num(v)$ und $\alpha(w) = 2$. Dann wird (v,w) Cross-Kante, denn es wurde ein Knoten erreicht, der schon komplett abgearbeitet ist. Der Knoten v kann desweiteren nicht direkt vom Knoten w erreicht werden, da (v,w), sonst eine Back-Kante wäre.

• **Wie funktioniert BFS?**
Bei der Tiefensuche (BFS=**B**readth-**F**irst-**S**earch), werden die Knoten in der Reihenfolge ihres Abstandes vom Startknoten aus erreicht. Dafür wird eine Queue benutzt.

Gegeben sei ein Graph als Knotenarray mit Adjazenzlisten. Am Anfang wird der Start-Knoten in die Queue gepackt. Dann wird seine Adjazenzliste durchlaufen und alle Knoten, die noch nicht in der Queue stehen, bzw. schon in ihr gestanden haben,

in sie geschrieben. Dann wird der Knoten ausgegeben und aus der Queue genommen. Dasselbe macht man mit allen anderen Knoten in der Queue, bis alle Knoten besucht wurden. Auch hier ist die Laufzeit $O(n+m) = O(|V| + |E|)$.

1.6 Union-Find-Datenstrukturen

- **Was sind Union-Find-Datenstrukturen?**

Union-Find-Datenstrukturen unterstützen folgende Operationen: Es seien n Objekte, o.B.d.A. mit $1, ..., n$ bezeichnet, gegeben. Zu Beginn seien diese in n disjunkte Mengen gegeben. Zwei Mengen können zusammengelegt werden und es kann gefragt werden in welcher Menge sich ein Objekt befindet.

1. FIND(x) mit $1 \leq x \leq n$: Gebe den Namen der Menge zurück, in der sich x befindet.

2. UNION(A,B,C): Vereinige die Mengen A und B zu einer Menge C. A und B werden dabei vernichtet, so daß FIND also immer eine eindeutige Antwort gibt.

- **Wieviele UNIONs kann es in einer Union-Find-Datenstrukturen höchstens geben?**

Es kann höchstens $(n-1)$ UNIONs geben, da die Anzahl der Mengen bei jedem UNION jeweils um eins verringert wird.

- **Was ist nötig, damit damit die Mengennamen in einem Array verwaltet werden können?**

Es ist nötig, daß die Mengennamen und die Objekte aus der Menge $\{1, ..., n\}$ sind.

- **Welche konkreten Datenstrukturen gibt es für Union-Find-Datenstruk-turen?**

Die Speicherung in einem Array mit Listen, die besonders FINDs unterstützt, sowie die Speicherung als wurzelgerichtete Bäume, die UNIONs besonders unterstützen.

- **Wie funktioniert die Speicherung in einem Array?**

Wir benutzen ein Array R der Länge n, wobei $R[i]$ jeweils den Namen der Menge enthält, in der sich i befindet. Als Namen der Menge können wir jeweils einen Repräsentanten aus der Menge wählen, z.B. für die Menge $\{2,4,6,8,9\}$ den Repräsentanten 6.

Wenn man jetzt jedoch $(n-1)$ UNIONs macht, muß man $(n-1)$-mal über das Array laufen und die Repräsentanten ändern, was zu einer Laufzeit von $(n-1)n = n^2 - n = \Theta(n^2)$ führt. Dies können wir jedoch vermeiden, indem wir für jede aktuelle Menge I eine Liste *LIST(I)* verwalten, in der die Objekte der Menge I stehen. Zusätzlich sei die Länge der jeweiligen Listen in einer Variablen *SIZE(I)* gespeichert. Wenn wir also ein UNION durchführen, müssen wir nur in der kürzeren Liste nachschauen, für welche Objekte wir die Repräsentanten ändern müssen und die beiden Listen aneinanderhängen.

Es kann auch vorkommen, daß zwischen internen und externen Namen unterschieden werden muß. Dann wird eine Lexikon angelegt, welches für die externen Namen *EXT(I)* die internen Namen *INT(I)* speichert. Im folgenden wird davon ausgegangen, daß auf *INT(I)* und *EXT(I)* in $O(1)$ zugegriffen werden kann. Also funktionieren FIND und UNION wie folgt:

1. FIND(x): FIND(x)=$EXT(R[x])$. Es wird also der Repräsentant von x gesucht und sein externer Name ausgegeben. Dies geht also in $O(1)$.

2. UNION(A,B,C):

   a) $I = INT(A), J = INT(B)$. Suche die entsprechenden internen Namen.

b) Ist $SIZE(I) \leq SIZE(J)$? Im folgenden wird davon ausgegangen.

c) Durchlaufe *LIST(I)* und für jedes $x \in LIST(I)$ setze $R[x] = J$. Ein Zeiger auf das letzte Element wird gemerkt.

d) Hänge die Liste $LIST(J)$ an $LIST(I)$ und benenne das Ergebnis in J um. $SIZE(J) = SIZE(I) + SIZE(J)$. Die Länge der neuen Liste ist also logischerweise die Summe der alten Listenlängen.

e) $INT(C) = J$ und $EXT(J) = C$. Das Lexikon wird aktualisiert.

Alle Schritte außer (3) brauchen Zeit $O(1)$. Schritt (3) beansprucht Zeit $O(SIZE(I))$.

- **In welcher Zeit kann eine Folge von $(n-1)$ UNIONs und f FINDs bei der Array-Speicherung ausgeführt werden?**

In Zeit $O(n\log_2 n + f)$. Es genügt zu zeigen, daß $(n-1)$ UNIONs in Zeit $O(n\log_2 n)$ ausgeführt werden können.

- **Zeige das $(n-1)$ UNIONs in $O(n\log_2 n)$ ausgeführt werden können!**

Die Anzahl der Operationen ist bis auf eine Konstante durch das Durchlaufen der jeweils kürzeren Liste beschränkt. Es wird $(n-1)$-mal die kürzere Liste durchlaufen. Uns interessiert also folgende Größe, wenn A_j die jeweils kürzere Liste ist:

$$\sum_{k=1}^{n-1} \sum_{x \in A_k} 1$$

Wir können uns diese Formel auch als eine $(n-1) \times n$-Matrix M vorstellen, für die $M(i,j) = 1$ ist, wenn sich das j-te Element in der Menge A_i befindet. Für die Gesamtlaufzeit müßten wir die Zeilensummen aufeinander addieren. Wir wissen jedoch nicht exakt, wie die Zeilensummen aussehen. Wir gehen jetzt anders an das Problem ran, indem wir einfach die einzelnen Spaltensummen abschätzen.

Die Spaltensumme der j-ten Spalte gibt an, wie oft das j-te Element bei einem UNION in der kleineren Menge ist. Wenn das j-te Element bei einem UNION in einer kleineren Menge der Größe r ist, ist es danach in einer Menge, deren Größe mindestens $2r$ beträgt, also mindestens doppelt so groß ist. Macht man dies s-mal, ist j daher in einer Menge mit mindestens 2^s Elementen. Die Größe der Menge ist jedoch nach oben durch n beschränkt. Daher gilt:

$$2^s \leq n \Leftrightarrow s \leq \lfloor \log_2 n \rfloor$$

Daraus folgt, daß j also höchstens $\lfloor \log_2 n \rfloor$-mal in der kleineren Menge sein kann. Also ist die Summe der einzelnen Spaltensummen durch $n\lfloor \log_2 n \rfloor$ beschränkt. Also durch $O(n\log_2 n)$.

Methoden, die nach dem Prinzip arbeiten, welches wir soeben angewandt haben, heißen auch Buchhaltermethode. Denn es war früher ein Buchhaltertrick, die Summe der Zeilensummen gegen die Summe der Spaltensummen noch einmal probezurechnen.

- **Wie funktioniert die Darstellung in wurzelgerichteten Bäumen?**

Die Darstellung in wurzelgerichteten Bäumen unterstützt besonders UNIONs. Es wird ein Array R der Länge n benutzt, wobei $R[i]$ den Elter des i-ten Elementes speichert.

Die Wurzeln zeigen dabei auf sich selbst, es gilt also $R[i] = i$. Desweiteren speichere jede Wurzel die Größe des Baumes, sowie den Namen der Menge.

- **Wie funktioniert ein UNION bei wurzelgerichteten Bäumen?**

UNION(A,B,C):

Schaue nach, ob A oder B kleiner ist. O.B.d.A. gelte im folgenden $SIZE(A) \leq SIZE(B)$. Setze als Elter der Wurzel von A die Wurzel von B. Desweiteren erhält die Wurzel des resultierenden Baumes den Namen C und $SIZE(C) = SIZE(A) + SIZE(B)$. Man hängt die kleinere Menge an die größere, damit die Bäume nicht allzu tief werden. Also sind UNIONs in konstanter Zeit durchführbar.

- **Wie funktionieren FINDs in der Darstellung durch wurzelgerichtete Bäume?**

FIND(x): Starte in x und laufe den eindeutigen Weg bis zur Wurzel. Die Laufzeit hierfür ist durch $O(\log_2 n)$ beschränkt.

- **Zeige, daß FIND durch $O(\log_2 n)$ beschränkt ist!**

Für jeden Knoten i gilt, daß seine Tiefe durch $O(\log_2 n)$ beschränkt ist. Die Argumentation ist ähnlich, wie bei UNION für die Array-Implementierung. Angenommen der Knoten i liegt in Tiefe d. Damit er in die Tiefe $(d+1)$ rutscht, müssen wir ein UNION ausführen, bei dem der Knoten i in der kleineren der beiden beteiligten Mengen liegt. Dadurch verdoppelt sich die Größe der resultierenden Menge jedoch mindestens. Um ein i also auf die Tiefe d zu bekommen, muß i also d-mal in der kleineren Menge sein und die Menge hat sich d-mal verdoppelt. Die Größe der Menge ist jedoch auch hier durch n beschränkt und es gilt:

$$2^d \leq n \Leftrightarrow d \leq \lfloor \log_2 n \rfloor$$

Daher kann ein Knoten i nur logarithmisch tief sein und bei FIND dementsprechend nur logarithmische Kosten verursachen.

- **Wie kann FIND bei wurzelgerichteten Bäumen verbessert werden?**

Durch „Path-Compression". Dabei werden alle Knoten, die bei einem FIND auf dem Suchpfad liegen direkt auf die Wurzel gerichtet. Die Kosten eines FIND wachsen dabei nur um einen konstanten Faktor. Zum Beispiel kann man die auf dem Weg liegenden Knoten auf einem Stack speichern, dessen Größe logischerweise auch logarithmisch beschränkt ist. Ist man an der Wurzel angekommen, richtet man alle Knoten, die auf dem Stack liegen dirket auf die Wurzel. Also kostet FIND $O(\log_2 n) + O(\log_2 n) = 2 \cdot O(\log_2 n) = O(\log_2 n)$. Zukünftige Suchen werden dabei jedoch unter Umständen enorm beschleunigt.

- **Welche Kosten verursachen n-1 UNIONs und f FINDs bei der Darstellung in wurzelgerichteten Bäumen mit Path-Compression?**

Die Kosten sind $O((n+f)\log_2^* n)$. Wobei $\log_2^*$ definiert ist als:

$$\log_2^* n = \min\{k | Z(k) \geq n\} \text{ mit } Z(k) = 2^{Z(k-1)} \text{ und } Z(0) = 1$$

Es gilt $\log_2^* n \leq 5$ für $n \leq 2^{65536} \approx 10^{19728}$. Für praktische Zwecke ist $\log_2^* n$ also quasi konstant, obwohl $\lim_{n\to\infty} \log_2^* n = \infty$ ist.

- **Gebe eine Anwendung für Union-Find-Datenstrukturen an!**

Ein Problem, bei dem man Union-Find-Datenstrukturen braucht, ist die Berechnung minimaler Spannbäume mit dem Algorithmus von Kruskal. Gegeben sei ein ungerichteter Graph $G = (V,E)$ mit Kantenbewertung $c : E \to \mathbb{R}^+$. Wir suchen einen Teilgraphen $G' = (V,E')$, der auf V zusammenhängend ist und unter allen Teilgraphen

mit dieser Eigenschaft minimale Kosten hat. Die Kosten eines solchen Graphen sind dabei:

$$\sum_{e \in E'} c(e)$$

Als Lösung kommt dabei logischerweise nur ein Baum auf V in Frage. Wir suchen also einen minimalen Spannbaum auf V.

Der Algorithmus von Kruskal berechnet minimale Spannbäume und beruht auf der Greedy-Methode.

- **Wie funktioniert der Algorithmus von Kruskal?**

Der Algorithmus von Kruskal arbeitet in vier Schritten:

1. Zusammenhang testen:

   Mit $DFS(G)$ wird getestet, ob G zusammenhängend ist. Ist G nicht zusammenhängend, gibt es keinen minimalen Spannbaum, sonst (2). Funktioniert in $O(n+m) = O(|V| + |E|)$.

2. Partitioniere:

   Bilde n einelementige Mengen $V_i = \{i\}, 1 \leq i \leq n$ mit Namen i für V_i. Weiter wird eine Variable N benutzt und es gilt hier: $N = |V|$, bzw. $N = n$, wenn o.B.d.A. gilt: $V = \{1, ..., n\}$. $T = \emptyset$, die Menge der Spannbaum-Kanten ist zu Beginn also leer. Geht in $O(n) = O(|V|)$.

3. Sortieren der Kanten:

   Sortiere die Kanten nach Kosten in aufsteigender Reihenfolge. Dieser Schritt geht in $O(m \log_2 m) = O(|E| \log_2 |E|)$.

4. Spannbaum aufbauen:

   Solange $N > 1$ ist, nehme die billigste noch nicht untersuchte Kante $e = (v, w)$. Verbindet die Kante e zwei unterscheidliche Partitionen, so nehme sie in die Menge der Tree-Kanten auf, sonst nehme die nächstbillige Kante. Wird e in die Menge der Tree-Kanten aufgenommen, so verschmelze die zugehörigen beiden Partitionen zu einer Partition. $N = N - 1$.

   Es müssen $(n-1)$ UNIONs gemacht werden, da ein Baum auf n Knoten genau $(n-1)$ Kanten hat. Desweiteren höchstens $2m = 2|E|$ FINDs. $2m$ FINDs werden gebraucht, wenn die letzte, also größte Kante, für den Spannbaum gebraucht wird.

1.7 Zusammenfassung

Problem	Laufzeit
Zugriff auf Arrayelement	$O(1)$
Vertauschen im Array	$O(1)$
Einfügen im Array	$\Omega(n)$
Platzbedarf UDM[1]	$O(n^2)$
Stackoperationen	$O(1)$
Queueoperationen	$O(1)$
Topologisches Sortieren naiv	$O(n^2+np)$
Topologisches Sortieren clever	$O(n+p)$
Addition zweier SPM_{list}[2]	$O(n+a+b)$
Hinzufügen, Entfernen, Query bei Bitvektor	$O(1)$
$\cup,\cap,-$ bei Bitvektor	$O(\|G\|)=O(n)$
Hinzufügen, Entfernen, Query bei UL[3]	$O(LENGHT(L))$
$\cup,\cap,-$ bei UL[3]	$O(LENGTH(L_1)\cdot LENGTH(L_2))$
$\cup,\cap,-$ bei OL[4]	$O(LENGTH(L_1)+LENGTH(L_2))$
DFS, BFS	$O(n+m)=O(\|V\|+\|E\|)$
$FIND_{array}$	$O(1)$
$UNION_{array}$	$O(LENGHT(KL^5))$
FIND in WGB[6]	$O(\log_2 n)$
UNION in WGB[6]	$O(1)$

[1] Untere Dreiecksmatrix
[2] Spärlich besetzte Matrizen in Listendarstellung
[3] Ungeornete Listen
[4] Geordnete Liste
[5] Kürzere Liste
[6] Wurzelgerichtete Bäume

2 Sortieralgorithmen

2.1 Einleitung

2.2 Insertion Sort

- **Wie funktioniert Insertion Sort?**

Insertion Sort funktioniert über binäre Suche. Man sucht in einem schon sortierten Array die Position, an der das neue Element stehen muß, und läßt alle Elemente rechts von dieser Position (inklusive dem Element, welches vorher dort stand) um eine Position nach rechts rücken.

Es seien i Daten sortiert, d.h. $a_1 \leq ... \leq a_i$. Es gibt $i+1$ Positionen an die das neue Element gehören kann. Denn es kann nach ganz vorne gehören (a_1) oder jeweils hinter eines der i schon vorhandenen Elemente. Es genügen $\lceil \log_2(i+1) \rceil$ Vergleiche, um die Position zu finden. Für das allererste Element wird kein Vergleich benötigt. Für das zweite wird schon einer benötigt ($\lceil \log_2 2 \rceil = 1$). Für das dritte werden im worstcase schon zwei benötigt ($\lceil \log_2 3 \rceil = \lceil 1.584... \rceil = 2$).

- **Wie viele Vergleiche benötigt Insertion Sort im worst case?**

Der worst case besteht z.B. darin, wenn eine schon aufsteigend bzw. absteigend sortierte Folge sortiert werden soll. In diesem Fall muß das neue Element nämlich jedesmal von der Mitte bis ganz zum Rand wandern. Da für das erste Element kein Vergleich gemacht werden muß, gilt für die Anzahl der Vergleiche:

$$\sum_{i=1}^{n-1} \lceil \log_2(i+1) \rceil = \sum_{i=2}^{n} \lceil \log_2(i) \rceil = \lceil \log_2 2 \rceil + \lceil \log_2 3 \rceil + ... + \lceil \log_2 i \rceil < \log_2(n!) + n$$

Wir untersuchen zunächst $n!$, um dann Aussagen über $\log_2(n!)$ machen zu können: Nach der Strirling'schen Formel gilt:

$$n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

Also gilt für $\log_2(n!)$:

$$\begin{aligned}
\log_2(n!) &\approx \log_2\left(\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n\right) \\
&= \log_2(\sqrt{2\pi}) + \log_2(\sqrt{n}) + n\log_2(n) - n\log_2 e \\
&= \underbrace{\overbrace{\log_2(\sqrt{2\pi})}^{konst.} + \frac{1}{2}\log_2(n)}_{O(\log_2(n))} + n\log_2(n) - n\underbrace{\log_2 e}_{1.4427...} \\
&= n\log_2(n) - 1.4427n + O(\log_2(n))
\end{aligned}$$

Für die Anzahl wesentlicher Vergleiche von Insertion Sort gilt also:

$$\begin{aligned}
\sum_{i=2}^{n} \lceil \log_2(i) \rceil &< n\log_2(n) - 1.4427n + O(\log_2(n)) + n \\
&= n\log_2(n) - 0.4427n + O(\log_2(n))
\end{aligned}$$

Also braucht Insertion Sort zum Sortieren von n Elemente höchstens $n\log_2(n) - 0.4427n + O(\log_2(n))$ wesentliche Vergleiche.

- **Wieviele Operationen (Rechtsverschiebungen) benötigt Insertion Sort im average case?**

Im worst case benötigt Insertion Sort $O(n^2)$ Operationen, wie man sich schnell klar machen kann: Zu Beginn, wenn das erste Element eingefügt wird, wird keine Verschiebung vorgenommen. Beim n-ten Element (das letzte Element) werden im worst case $n-1$ andere Elemente verschoben. Also gilt hier:

$\sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i = \frac{1}{2}(n-1)n = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$

Doch wie sieht es im average case aus? Wie sich herausstellen wird, kommt Insertion Sort hier mit $\Theta(n^2)$ Operationen aus.

Es gibt $(n-1)!$ verschiedene Permutationen π mit $\pi(i) = j$, d.h. daß das Element x_i an Position j stehen muß. Dort muß es mit Wahrscheinlichkeit $\frac{1}{n}$ stehen, da es insgesamt $n \cdot (n-1)! = n!$ Permutationen gibt. Die $(n-1)!$-Permutationen sind also gleichverteilt.

Das Element x_i muß also, wie gesagt, mit Wahrscheinlichkeit $\frac{1}{n}$ an Position j stehen. Dabei liegt j zwischen 1 und n, also $1 \leq j \leq n$. Interessant ist jedoch nur die Anzahl an Rechtsbewegungen, d.h. die Fälle $j > i$. Daraus ergibt sich der average case für ein x_i:

$$\begin{aligned}
\frac{1}{n}(\underbrace{0+\ldots+0}_{j<i} + \underbrace{0}_{j=i} + \underbrace{1}_{j=i+1} + \ldots + \underbrace{(n-i)}_{j=n}) &= \frac{1}{n} \cdot \sum_{k=0}^{n-i} k \\
&= \frac{1}{n} \cdot \frac{1}{2}(n-i)(n-i+1) \\
&= \frac{(n-i)(n-i+1)}{2n}
\end{aligned}$$

$\frac{(n-i)(n-i+1)}{2n}$ ist also die durchschnittliche Anzahl an Rechtsbewegungen, die ein beliebiges Element x_i macht, wenn es eigentlich an Position j gehört.

Für den gesamten average case muß man i von 1 bis n wandern lassen und diese Werte aufsummieren. Also:

$$
\begin{aligned}
\sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2n} &= \frac{1}{2n}\sum_{i=1}^{n}(n-i)(n-i+1) \\
&= \frac{1}{2n}\sum_{i=1}^{n}(i-1)i \\
&= \frac{1}{2n}\sum_{i=1}^{n}(i^2-i) \\
&= \frac{1}{2n}\sum_{i=1}^{n} i^2 - \frac{1}{2n}\sum_{i=1}^{n} i \\
&= \frac{1}{2n}\frac{1}{6}n(n+1)(2n+1) - \frac{1}{2n}\frac{1}{2}n(n+1) \\
&= \frac{1}{12}(n+1)(2n+1) - \frac{1}{4}(n+1) \\
&= \frac{1}{12}(2n^2+n+2n+1) - \frac{1}{4}n - \frac{1}{4} \\
&= \frac{1}{6}n^2 + \frac{1}{4}n + \frac{1}{12} - \frac{1}{4}n - \frac{1}{4} \\
&= \frac{1}{6}n^2 - \frac{1}{6} = \frac{1}{6}(n^2-1) \\
&= \Theta(n^2)
\end{aligned}
$$

2.3 Quick Sort

Quick Sort wurde 1962 von Hoare vorgestellt und ist das heute am häufigsten benutzte interne Sortierverfahren. Es arbeitet in situ (wenn der Rekursionsstack durch $O(\log_2 n)$ beschränkt ist).

- **Wie funktioniert Quick Sort?**

Gegeben sei ein Array. Wähle ein Datum a_i aus. Hierzu gibt es verschiedene Methoden:

1. Wähle immer das an erster Position stehende Datum.
2. Wähle mit einem Zufallszahlengenerator zufällig ein Element aus dem Array aus.
3. Wähle im Array drei bestimmte Daten. Und zwar das erste, das letzte und das in der Mitte ($\lceil \frac{n}{2} \rceil$, bei n Elementen) stehende Element. Berechne das bezüglich der Größe mittlere dieser drei und nehme dies.
4. Wähle mit einem Zufallsgenerator drei Daten und bestimme deren Median.

Ist ein Datum gewählt, sorge dafür daß es an die richtige Position kommt und links davon nur kleinere Daten und rechts davon nur größere Daten stehen. Danach wende Quick Sort rekursiv auf das linke und rechte Teilarray an. Für Arrays der Länge 0 oder 1 kann gestoppt werden.

- **Was muß beachtet werden, damit Quick Sort in situ arbeitet?**

Damit ein Verfahren in situ arbeitet, muß man dafür sorgen, daß der zusätzlich gebrauchte Speicher in seiner Größe durch $O(\log_2 n)$ beschränkt ist. Dies ist gewährleistet, wenn man immer zuerst das kleinere Teilarray bearbeitet.

- **Wie ist die worst case-Laufzeit von Quick Sort?**

Die worst case-Laufzeit beträgt $\Theta(n^2)$. Hier unterscheiden sich die Auswahlmethoden 1 und 2 nur geringfügig von den Methoden 3 und 4.

Bei den Varianten 1 und 2 kann das gewählte Element das kleinste (größte) Element der Menge sein. Also gilt für die Anzahl der Vergleiche, daß man mindestens $(n-1)$ Vergleiche macht plus die Vergleiche des Restarrays der Größe $(n-1)$:

$$\begin{aligned} V_{wc}(n) &\geq V_{wc}(n-1) + (n-1) \\ &\geq 1 + 2 + \ldots + (n-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{1}{2}(n-1)n \\ &= \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2) \end{aligned}$$

Bei den Varianten 3 und 4 ist es etwas günstiger, jedoch nicht sehr viel. Denn auch hier ist die worst case-Laufzeit $\Theta(n^2)$. Hier kann das gewählte Element nicht das kleinste (größte) sein, da man ja den Median von drei Daten berechnet. Die Berechnung des Medians benötigt im worst case jedoch schon drei Vergleiche (z.B.

$x_1x_2x_3 : x_1 \leq x_2?, x_2 \leq x_3?, x_1 \leq x_3?$). Dafür muß man im worst case nur ein Array der Teilgröße $(n-2)$ bearbeiten, da ein Teilarray in diesem Fall die Größe 1 hat, also 0 Vergleiche braucht. Wir schätzen $V_{wc}(n)$ nach unten grob ab durch:

$$\begin{aligned}
V_{wc}(n) &\geq V_{wc}(n-2)+n \\
&\geq n+(n-2)+(n-4)+...+(n-n) \\
&= \underbrace{n+n+...+n}_{\frac{1}{2}n\cdot n} - (0+2+4+...+n) \\
&= \frac{1}{2}n\cdot n - \sum_{i=0}^{\frac{1}{2}n} 2i \\
&= \frac{1}{2}n^2 - 2\sum_{i=0}^{\frac{1}{2}n} i \\
&= \frac{1}{2}n^2 - 2\cdot\frac{1}{2}\frac{1}{2}n(\frac{1}{2}n+1) \\
&= \frac{1}{2}n^2 - (\frac{1}{4}n^2+\frac{1}{2}n) \\
&= \frac{1}{4}n^2 - \frac{1}{2}n \\
&= \Theta(n^2)
\end{aligned}$$

- **Wie ist die average case-Laufzeit von Quick Sort?**

Beim average case gehen wir davon aus, daß jedes Element mit gleicher Wahrscheinlichkeit $\frac{1}{n}$ gewählt wird. Wird ein Element gewählt, welches nach $n-1$ Vergleichen an Position i steht, so stehen links davon $i-1$ Elemente und rechts davon $n-i$ Elemente, die rekursiv mit Quick Sort sortiert werden. Daraus ergibt sich:

$$\begin{aligned}
V(n) &= n-1+\frac{1}{n}\sum_{i=1}^{n}(V(i-1)+V(n-i)) \\
V(n) &= n-1+\frac{2}{n}\sum_{i=1}^{n}V(i-1) \\
nV(n) &= n(n-1)+2\sum_{i=1}^{n}V(i-1)
\end{aligned}$$

Jetzt wird jedes n durch $(n-1)$ in der Gleichung ersetzt.

$$(n-1)V(n-1) = (n-1)(n-2)+2\sum_{i=1}^{n-1}V(i-1)$$

Jetzt subtrahieren wir $V(n-1)$ von $V(n)$, um die Summe loszuwerden.

$$\begin{aligned}
nV(n)-(n-1)V(n-1) &= \underbrace{n(n-1)-(n-1)(n-2)}_{2(n-1)}+2V(n-1) \\
nV(n)-(n+1)V(n-1) &= 2(n-1)
\end{aligned}$$

Division durch $n(n+1)$.

$$\begin{aligned} \frac{V(n)}{n+1} - \frac{V(n-1)}{n} &= \frac{2(n-1)}{n(n+1)} \\ \frac{V(n)}{n+1} &= \frac{V(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \end{aligned}$$

Wir setzen $Z(n) = \frac{V(n)}{n+1}$.

$$\begin{aligned} Z(n) &= Z(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= Z(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= Z(1) + 2\sum_{i=2}^{n} \frac{i-1}{i(i+1)} \end{aligned}$$

Da $Z(1) = 0$ ist, gilt also:

$$Z(n) = 2\sum_{i=2}^{n} \frac{i-1}{i(i+1)}$$

Es gilt desweiteren:

$$\frac{1}{i(i+1)} = \frac{1}{i} - \frac{1}{i+1}$$

Dies führt zu folgender Formel:

$$\begin{aligned}
2\sum_{i=2}^{n}\left(\frac{i-1}{i}-\frac{i-1}{i+1}\right) &= 2\sum_{i=2}^{n}\frac{i-1}{i}-2\sum_{i=2}^{n}\frac{i-1}{i+1}\\
&= 2\left(\sum_{i=3}^{n}\frac{i-1}{i}+\frac{1}{2}\right)-2\sum_{i=3}^{n+1}\frac{i-2}{i}\\
&= 1+2\sum_{i=3}^{n}\frac{i-1}{i}-2\left(\sum_{i=3}^{n}\frac{i-2}{i}+\frac{n-1}{n+1}\right)\\
&= 1+2\sum_{i=3}^{n}\frac{i-1}{i}-2\sum_{i=3}^{n}\frac{i-2}{i}-2\frac{n-1}{n+1}\\
&= 1+2\left(\sum_{i=3}^{n}\frac{i-1}{i}-\sum_{i=3}^{n}\frac{i-2}{i}\right)-2\frac{n-1}{n+1}\\
&= 1+2\left(\sum_{i=3}^{n}\left(\frac{i-1}{i}-\frac{i-2}{i}\right)\right)-2\frac{n-1}{n+1}\\
&= 1+2\left(\sum_{i=3}^{n}\left(\frac{i-1-i+2}{i}\right)\right)-2\frac{n-1}{n+1}\\
&= 1+2\sum_{i=3}^{n}\frac{1}{i}-2\frac{n-1}{n+1}\\
&= 1+2\sum_{i=1}^{n}\frac{1}{i}-\underbrace{2\cdot(\frac{1}{1}+\frac{1}{2})}_{3}-2\frac{n-1}{n+1}\\
&= 2\sum_{i=1}^{n}\frac{1}{i}-2\frac{n-1}{n+1}-2
\end{aligned}$$

Die Folge $\sum_{i=1}^{n}\frac{1}{i}$ kommt so häufig vor, daß sie einen eigenen Namen bekommen hat, nämlich Harmonische Reihe $H(n)$. Es gilt also:

$$Z(n)=2H(n)-2\frac{n-1}{n+1}-2$$

Wenn $Z(n)=\frac{V(n)}{n+1}$ ist, gilt $V(n)=Z(n)\cdot(n+1)$. Also:

$$\begin{aligned}
V(n) &= 2H(n)(n+1)-2\frac{n-1}{n+1}(n+1)-2(n+1)\\
&= 2H(n)(n+1)-2(n-1)-2(n+1)\\
&= 2H(n)(n+1)-2n+2-2n-2\\
&= 2(n+1)H(n)-4n
\end{aligned}$$

Aus der Betrachtung Riemannscher Summen folgt:

$$\int_{1}^{n+1}\frac{1}{x}dx\le H(n)\le 1+\int_{1}^{n}\frac{1}{x}dx$$

Und damit folgt weiter:

$$\ln(n+1) \leq H(n) \leq 1+\ln(n)$$

Es läßt sich sogar zeigen, daß $H(n)-\ln(n)$ konvergiert und zwar gegen die Eulersche Konstante $\gamma \approx 0,57721...$. Wenn also $H(n)-\ln(n)=\gamma$ gilt, so gilt auch $H(n)=\ln(n)+\gamma$. Wir ersetzen in der Formel also $H(n)$ durch $\ln(n)+\gamma$:

$$\begin{aligned} V(n) &= 2(n+1)H(n)-4n \\ &\approx 2(n+1)(\ln(n)+0.57721)-4n \end{aligned}$$

Desweiteren gilt:

$$\log_2(n)=\frac{\ln(n)}{\ln(2)}, \text{ also } \ln(n)=\log_2(n)\cdot\ln 2$$

$$\begin{aligned} V(n) &\approx 2(n+1)(\ln(n)+0.57721)-4n \\ &= (2n+2)(\log_2(n)\cdot\ln 2+0.57721)-4n \\ &= 2\cdot\ln 2\cdot n\log_2(n)+1.15442n+2\cdot\ln 2\cdot n+1.15442-4n \\ &= 1.386\cdot n\log_2(n)-2.846n+1.386\cdot\log_2(n)+1.154 \end{aligned}$$

Zusammenfassend läßt sich also sagen, daß alle Quicksort-Varianten eine worst case-Laufzeit von $\Theta(n^2)$ haben. In der average case-Laufzeit unterscheiden sich die Varianten 1 und 2 von den Varianten 3 und 4.
Die Varianten 1 und 2 haben eine average case-Laufzeit von:

$$2(n+1)H(n)-4n \approx 1.386n\log_2(n)-2.846n+1.386\log_2 n+1.154$$

Die Analyse der Varianten 3 und 4 ist wesentlich schwieriger. Die Zahl der wesentlichen vergleiche beträgt für $n \geq 6$ im Durchschnitt:

$$\frac{12}{7}(n+1)H(n-1)-\frac{477}{147}n+\frac{223}{147}+\frac{252}{147n} \approx$$

$$1.188n\log_2(n-1)-2.255n+1.188\log(n-1)+2.507$$

2.4 Heap Sort und Bottom-Up-Heap Sort

Heap Sort wurde 1964 in verschiedenen Varianten von Williams vorgestellt. Dieses Sortierverfahren arbeitet in situ, ist jedoch nur mit zusätzlichem Aufwand zu stabilisieren.

- **Auf welcher Datenstruktur arbeitet Heap Sort und wie wird diese interpretiert?**

Heap Sort arbeitet auf einem Array, welches als ein binärer Baum interpretiert wird. Es gibt insbesondere keine Zeiger.

- **Wie kann man die Kinder eines Knotens finden, wenn es keine Zeiger gibt?**

Position 1 ist die Wurzel des Baumes. Ist i ein beliebiger Knoten des Baumes, so steht sein linker Sohn an Position $2i$, sofern $2i \leq n$. n ist hierbei die Länge des Arrays,

also die Gesamtzahl der Knoten. Sein rechter Sohn befindet sich an Position $(2i+1)$, sofern $(2i+1) \leq n$.

Die Eltern befinden sich dementsprechend an Position $\lfloor \frac{i}{2} \rfloor$, für $i \geq 2$. Der Vorgänger im Abstand k befindet sich an Position $\lfloor \frac{i}{2^k} \rfloor$, falls $i \geq 2^k$. Dabei sollte man jedoch Shift-Operationen benutzen, da diese schneller sind als normale Divisionen.

- **Wann ist ein Array ein Heap?**

Ein Array heißt Min-Heap (Max-Heap), wenn das Datum an jedem Knoten kleiner (größer) ist als das seiner beiden Kinder (falls existent).

- **Wie arbeitet Heap Sort?**

Heap Sort läßt sich grob in zwei Phasen einteilen. Der Heap Creation Phase und der Selection Phase.

In der Heap Creation Phase wird aus dem Array ein Heap gemacht. Dabei wird mit Hilfe der Methode *reheap* an den Knoten $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{2} \rfloor - 1, ..., 1$ in dieser Reihenfolge die Heapbedingung hergestellt. Nach dieser Phase steht das kleinste Datum der Gesamtmenge an Position 1, also an der Wurzel des Heaps.

Anschließend arbeitet Heap Sort nur noch in der Selection Phase. Zuerst wird das erste mit dem letzten Element vertauscht und danach *reheap*$(1, n-1)$ aufgerufen. Danach steht wieder das kleinste Element dieser Menge ganz oben und es wird mit dem Element an Position $(n-1)$ vertauscht.

Das Rahmenprogramm sieht also für ein Array der Länge n wie folgt aus:

1. Heap Creation Phase:

```
for(int i=n/2;i>0;i--)
  reheap(i,n);
```

2. Selection Phase:

```
for(int m=n;m>1;m--)
  {
    tausch(a[1],a[m]);
    reheap(1,m-1);
  };
```

Einmal wird *reheap* also zum Aufbau eines Heaps eingesetzt (Heap Creation) und dann nur noch zum reparieren (Selection Phase).

- **Wie sieht die Methode** *reheap*(i,m) **aus?**

Es gibt verschiedene Versionen von *reheap*. Hier wird jetzt die alte Version beschrieben, eine andere effizientere folgt noch. Die alte Methode verschiebt das Problem von der Wurzel des betrachteten Teilbaums Ebene für Ebene, bis kein Problem mehr besteht. Sie läßt einen Knoten also an seine richtige Position im Heap einsinken. Dabei unterscheidet man drei Fälle:

```
if(i>m/2) STOP
if(i=m/2)
  {
    if(a[i] <= a[2i] STOP
    else { tausch(a[i],a[2i]; STOP;};
```

```
  };
if(i<m/2)
  {
    if a[2i]<=a[2i+1] l=2i
    else l=2i+1
    if(a[i]<=a[l] STOP
    else { tausch(a[i],a[l]); reheap(l,m);};
  };
```

Im ersten Fall ($i > \frac{m}{2}$) besteht kein Problem, da wir an einem Blatt sind. In Blättern ist die Heap-Bedingung trivialerweise erfüllt.

Im zweiten Fall ($i = \frac{m}{2}$) gibt es nur ein Kind. Existiert hier ein Problem, wir es behoben und wir können stoppen, da wir an einem Blatt fortsetzen würden.

Im dritten Fall ($i < \frac{m}{2}$) gibt es zwei Kinder. Wir suchen das kleinere auf und vergleichen das Datum an der Wurzel mit diesem Kind. Ist das Datum an der Wurzel kleiner, so ist alles in Ordnung. Sonst wird das Problem um eine Ebene nach unten geschoben, indem die Daten vertauscht werden und *reheap* mit dem Kind als Wurzel wieder aufgerufen wird.

- **Wieviele Vergleiche werden bei <u>einem</u> Aufruf der Methode reheap höchstens gemacht?**

Sei d die Tiefe des Teilbaumes mit der Wurzel, für die reheap aufgerufen wurde. Dann werden höchstens $2d$ Vergleiche gemacht, da in jedem Aufruf von reheap höchstens zwei Vergleiche gemacht werden.

- **Wieviele wesentliche Vergleiche werden in der Heap Creation Phase höchstens gemacht?**

Dazu betrachten wir erst einmal, wie oft *reheap* während dieser Phase aufgerufen wird. Hat ein Teilbaum die Tiefe d so wird *reheap* höchstens d-mal aufgerufen. Also ist die Gesamtzahl an Aufrufen gleich der Summe der Tiefen aller Knoten für die *reheap* aufgerufen wird.

Es wird die Methode für $\lfloor \frac{n}{2} \rfloor$ Bäume mit Wurzeln $1,...,\lfloor \frac{n}{2} \rfloor$ aufgerufen. Davon sind nur die Knoten $1,...,\lfloor \frac{n}{4} \rfloor$ Wurzeln von Bäumen, deren Tiefe mindestens 2 beträgt. Und nur die Knoten $1,...,\lfloor \frac{n}{8} \rfloor$ sind Wurzeln von Bäumen deren Tiefe mindestens 3 beträgt.

Allgemein gilt also: Nur die Knoten $1,...,\lfloor \frac{n}{2^d} \rfloor$ sind Wurzeln von Bäumen deren Tiefe mindestens d beträgt. Wir summieren diese Anzahlen für $1 \leq d \leq \lfloor \log_2(n) \rfloor$ auf. Dabei zählen wir die Wurzel z.B. d-mal. Die Kinder der Wurzel $(d-1)$-mal.

$$\sum_{d=1}^{\lfloor \log_2(n) \rfloor} \frac{n}{2^d} = n \sum_{d=1}^{\lfloor \log_2(n) \rfloor} \frac{1}{2^d} < n \sum_{d=1}^{\infty} \frac{1}{2^d} = n \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... \right) = n \cdot 1 = n$$

Also wir die Methode *reheap* weniger als n-mal aufgerufen. Da jeder Aufruf von *reheap* höchstens 2 Vergleiche macht, werden also insgesamt weniger als 2n Vergleiche gemacht. Daraus folgt, daß die Heap Creation Phase Zeit $O(n)$ braucht.
Es gibt noch einem alternativen Beweis:

Sei B ein vollständiger Baum mit n Elementen. Dann haben $\frac{n}{4}$ Knoten (Teilbäume) die Tiefe 1, nämlich die „vorletzte Reihe". Dies ergibt sich aus $\frac{n}{2} - \frac{n}{4} = \frac{n}{4}$.

$\frac{n}{2}$ ist hierbei die Anzahl der Knoten des Baumes, wenn wir die Blätter abziehen und $\frac{n}{4}$ ist die Anzahl der Knoten des Baumes, ohne die Blätter und die Elemente die

wir zählen wollen. Dies kann man beliebig fortsetzen, d.h. $\frac{n}{8}$ Knoten haben die Höhe 2, $\frac{n}{16}$ Knoten die Höhe 3, etc.
Hieraus ergibt sich:

$$\frac{n}{4}\cdot 1+\frac{n}{8}\cdot 2+\frac{n}{16}\cdot 3+...<\frac{n}{2}\underbrace{\sum_{i=0}^{\infty}\frac{i}{2^i}}_{2}=\frac{n}{2}\cdot 2=n$$

- **Wieviele Vergleiche benötigt man höchstens in der Selection Phase?**

Zuerst betrachten wir die Anzahl an Aufrufen, die in der Selection Phase stattfinden. Der Gesamtbaum hat Tiefe $\lfloor \log_2(n) \rfloor$. Reheap wird $(n-1)$-mal aufgerufen. Also ist $n \cdot \log_2(n)$ eine obere Schranke. Wieder braucht jeder Aufruf von *reheap* höchstens 2 Vergleiche. Also ist $2 \cdot n \cdot \log_2(n)$ eine obere Schranke.

- **Wieviele Vergleiche macht der Standard Heap Sort im worst case?**

$2 \cdot n \cdot \log_2(n) + 2n$. $2n$ für den Aufbau und $2 \cdot n \cdot \log_2(n)$ in der Selection Phase.

- **Wieviele wesentliche Vergleiche macht der Standard Heap Sort im average case?**

Im Durchschnitt über alle Permutationen werden nach Munro $2 \cdot n\log_2(n) - O(n)$ Vergleiche durchgeführt.

- **Wie kommt es zu dem Faktor 2 und was kann man dagegen machen?**

Der Faktor 2 kommt dadurch zustande, daß in der Selection Phase ein Datum, welches ziemlich groß ist, da es ja vorher ein Blatt war und damit ziemlich weit unten stand, nach oben getauscht wird und dann wieder eingesunken wird. Dabei wandert es sehr oft wieder ziemlich weit nach unten. Daher werden in der Nähe der Wurzel viele „unnütze" Vergleiche gemacht.

Um dies zu verhindern, kann man mit je einem Vergleich pro Knoten den Weg bestimmen, wo das Datum langwandern würde (es würde ja immer am kleineren Knoten weiterwandern) und auf diesem Weg die entgültige Position des Datum intelligenter bestimmen. Dies führt zu *Bottom-Up-reheap*, welcher *reheap* ersetzen kann.

- **Wie funktioniert Bottom-Up-reheap?**

Grob kann man Bottom-Up-reheap in drei Teile aufteilen.

1. leaf-search
2. Bottom-Up-search
3. interchange

Leaf-search sucht das Blatt, bis zu dem das Datum an der Wurzel höchstens wieder wandern würde. Bottom-Up-search sucht auf dem so gefundenen Weg die Position, an die das Datum hinpaßt und interchange führt die Datentransporte durch.

- **Wie funktioniert leaf-search?**

Leaf-search bestimmt ein Blatt und damit einen Weg, auf dem das getauschte Datum an der Wurzel eingefügt werden muß. Folgender Algorithmus:

```
int leaf_search(int i, int m)
       {
          int j=1;
          while(2j<m)
```

```
        {
           if a[2j]<a[2j+1] j=2j
           else j=2j+1
        };
      if 2j=m j=m;
      return j;
    };
```

Wenn man das Blatt j kennt, kennt man auch den gesamten Weg von der Wurzel bis zum Blatt j. Man kann jedoch auch gleich die Knoten bis zum Blatt j explizit abspeichern, ohne die in situ-Bedingung zu verletzen. Ein Weg kann nämlich höchstens $\log_2(n)$ haben. Also ist der Platzbedarf zur Speicherung der Knoten $O(\log_2(n))$.

- **Wieviele wesentliche Vergleiche macht leaf-search höchstens?**

In jedem Aufruf verursacht leaf-search einen Vergleich. In der Heap Creation Phase wird leaf-search weniger als n-mal aufgerufen und in der Selection Phase höchstens $n\log_2(n)$. Also ist $n\log_2(n)+n$ eine obere Schranke für die Vergleiche.

- **Wie arbeitet Bottom-Up-search? Welche Varianten gibt es?**

Sei x das Datum an der Wurzel und $y_1,...,y_l$ die Daten auf dem Weg zum Blatt. Nach der Vertauschung ist die Heap-Bedingung höchstens an der Wurzel verletzt.

Die alte reheap-Methode hat von der Wurzel aus das kleinste k mit $x \leq y_k$ gesucht. Die Daten $y_1,...,y_{k-1}$ wandern an die Elter-Position und x kommt an die Position x_{k-1}. Da gleiche Daten erlaubt sind, ist also jede Position $y_{k-1} \leq x \leq y_k$ erlaubt.

Wie schon gesagt, liegt diese y_k oft nicht sehr nah bei der Wurzel. Es gibt jetzt zwei andere Möglichkeiten dieses Datum y_k zu finden.

Zum Beispiel kann man die entsprechende Position mit binärer Suche bestimmen. Im worst case braucht man bei einer binären Suche über i sortierte Elemente Zeit $\lceil\log_2(i+1)\rceil$ (siehe Insertion Sort). i ist in diesem Fall die maximale Anzahl an Knoten, die auf dem Weg zum Blatt j liegen. Diese Anzahl ist $\lfloor\log_2(n)\rfloor$, also gilt für die wesentlichen Vergleiche, daß sie durch $\lceil\log_2(\lfloor\log_2(n)\rfloor+1)\rceil \leq \log_2(\log_2(n))+1$ beschränkt sind.

Es zeigt sich jedoch, daß eine lineare Suche, die von Blatt aus startet cleverer ist. Dies kommt daher, daß das Datum an der Wurzel, welches durch eine Vertauschung von der letzten Position dort hin kam, relativ groß ist. Denn es war ja ein Blatt und hat unter Umständen viele Vorgänger die kleiner sind. Das Blatt, welches wir mit leaf-search finden, ist jedoch relativ klein, da wir bei leaf-search ja immer am kleineren Kind weitergehen. In Simulationen hat sich gezeigt, daß das Datum im Blatt in 85% der Fälle kleiner ist als das Datum in der Wurzel. Also kann man lieber am Blatt starten und unter Umständen schnell merken, daß das Datum in der Wurzel wieder in ein Blatt muß bzw. sehr weit runterwandern muß. Dies führt zu folgendem Algorithmus:

```
int bottom_up_search(int i, int j)
      {
        while(a[i]<a[j]) j=j/2;
        return j;
      };
```

Die Schleife bricht für i=j ab, d.h. wenn wir an der Wurzel sind. Mit einem unwesentlichen Vergleich können wir den Vergleich der Wurzel mit sich selbst unterbinden ($i < j$?).

- **Wie funktioniert interchange?**

Angenommen der Weg von der Wurzel bis zum Blatt j ist in einem Array $b(0) = i, ..., b(l) = j$ abgespeichert. Dann führt folgender Algorithmus den Datentransport durch:

```
{
  x=a[i];
  for(int k=1; k<=l;k++)
    {
      a[b[k-1]]=a[b[k]];
    };
  a[j]=x;
};
```

Das es höchstens $\log_2(n)$ Knoten gibt die verschoben werden müssen, arbeitet dieser Algorithmus in Zeit $O(\log_2(n))$.

- **Welche worst case-Anzahl von Vergleichen hat die Bottom-Up-Heap Sort Variante, die in der Heap Creation Phase das alte reheap benutzt und in der Selection Phase mit binärer Suche arbeitet?**

Die Vergleiche in der Heap Creation Phase sind durch $2n$ beschränkt. Wir rufen n mal leaf-search auf, dessen wesentliche Vergleiche durch $n\log_2(n)$ beschränkt sind. Eine obere Schranke für die Vergleiche von Bottom-Up-search mit binärer Suche ist $n(\log_2(\log_2(n)) + 1)$, da wir höchstens n-mal Bottom-Up-reheap aufrufen. Daraus ergibt sich:

$$V_{wc} = n\log_2(n) + n(\log_2(\log_2(n)) + 1) + 2n = n\log_2(n) + n\log_2(\log_2(n)) + 3n$$

Im average case ist diese Variante jedoch uninteressant.

- **Welche average case-Rechenzeit hat Buttom-Up-Heap Sort mit linearer Suche?**

Die average case-Rechenzeit über alle Permutationen beträgt $n\log_2(n) + O(n)$ für die wesentlichen Vergleiche und $O(n\log_2(n)$ für die anderen Operationen (Munro).

- **Zeige, daß beim Bottom-Up-Heap Sort mit linearer Suche kein worst-case von $2n\log_2(n) + O(n)$ Vergleichen mehr droht!**

Man benötigt im worst case höchstens $1.5n\log_2 n + O(n)$ wesentliche Vergleiche. Leaf-Search benötigt höchstens $n\log_2 n + n$ wesentliche Vergleiche und Bottom-Up-Search in der Heap Creation Phase höchstens n. Zu zeigen ist also, daß Bottom-Up-Search nur noch Kosten $0.5n\log_2 n + O(n)$ verursacht.

Wir gehen im folgenden davon aus, daß $n = 2^k$ ist und unterteilen die Daten in große und kleine Daten. Ein Datum heißt groß, wenn es zur grösseren Hälfte aller Daten gehört, sonst heißt es klein. Die Hälfte aller Daten, das sind also 2^{k-1}, stehen an den Blättern und da große Daten große Nachfolger haben, ist mindestens die Hälfte aller Daten an den Blättern groß. Wir können also davon ausgehen, daß 2^{k-2} Blätter groß sind und 2^{k-2} Daten klein. Insgesamt wollen wir jetzt zeigen, daß die ersten $2^{k-1} = 0.5n$ Aufrufe von Bottom-Up-Search höchstens Kosten $0.25n\log_2 n + O(n)$ Kosten verursachen. Danach haben wir ein Array halber Größe und können analog argumentieren.

Für die kleinen Daten setzen wir als worst case $(\log_2 n - 1)$ Vergleiche an, da sie im worst case wieder bis ganz an die Wurzel wandern können, was für große Daten logischerweise nicht der Fall ist (sonst wäre es kein großes Datum). $2^{k-2} = 0.25n$ Daten sind klein, also verursachen sie Kosten $2^{k-2}(\log_2 n - 1) = 0.25\log_2 n - 0.25n$. Alles was wir jetzt noch zeigen müssen, ist daß die 2^{k-2} großen Daten nur noch linearen Aufwand benötigen.

Es sei $d(m)$ die Ebene auf der das Datum an der Wurzel beim m-ten Aufruf von Bottom-Up-Search landet. Dann braucht Bottom-Up-Search bis es die Position findet höchstens $k - d(m)$ Vergleiche. Das Schlimmste, was uns passieren kann, ist daß die großen Daten alle ganz oben im Heap eingeordnet werden, da in diesem Fall die meisten Vergleiche gebraucht werden. Uns interessiert also folgender Wert:

$$\sum_{m=1}^{2^{k-2}} (k - d(m)) = \sum_{m=1}^{2^{k-2}} k - \sum_{m=1}^{2^{k-2}} d(m) = 2^{k-2}k - \sum_{m=1}^{2^{k-2}} d(m)$$

Letzterer Wert ist für den worst case die Summe der Tiefen für 2^{k-2} Knoten in einem binären Baum. Der Baum hat also die Tiefe $k-2$ und es gilt:

$$\sum_{i=1}^{k-2} i2^i \geq \sum_{i=1}^{k-3} i2^i + (k-2)$$

Betrachten wir allgemein $\sum_{i=1}^{n} i2^i$:

$$\begin{aligned}
\sum_{i=1}^{n} i2^i &= 1\cdot 2^1 + 2\cdot 2^2 + 3\cdot 2^3 + \ldots + n\cdot 2^n \\
&= 2^1 + 2^2 + 2^3 + \ldots + 2^n \\
&\quad\; + 2^2 + 2^3 + \ldots + 2^n \\
&\quad\; + 2^3 + \ldots + 2^n \\
&\quad\; \ldots \\
&\quad\; + 2^n \\
&= (2^{n+1} - 1 - (2^1 - 1)) \\
&\quad + (2^{n+1} - 1 - (2^2 - 1)) \\
&\quad + \ldots \\
&\quad + (2^{n+1} - 1 - (2^n - 1)) \\
&= n2^{n+1} - \sum_{i=1}^{n} 2^i \\
&= n2^{n+1} - (\sum_{i=0}^{n} 2^i - 1) \\
&= n2^{n+1} - ((2^{n+1} - 1) - 1) \\
&= (n-1)2^{n+1} + 2
\end{aligned}$$

Demnach gilt:

$$\sum_{i=1}^{k-3} i2^i = (k-4)2^{k-2} + 2$$

Wir setzen dieses Ergebnis oben ein und schätzen weiter ab:

$$\sum_{i=1}^{k-2} i2^i \geq \sum_{i=1}^{k-3} i2^i + (k-2) = (k-4)2^{k-2} + 2 + (k-2) \geq (k-4)2^{k-2}$$

Also ergibt sich insgesamt:

$$\begin{aligned}
\sum_{m=1}^{2^{k-2}} (k - d(m)) &= 2^{k-2}k - (k-4)2^{k-2} \\
&= 2^{k-2}k - 2^{k-2}k + 4 \cdot 2^{k-2} \\
&= 2^2 \cdot 2^{k-2} \\
&= 2^k \\
&= n
\end{aligned}$$

2.5 Merge Sort

Merge Sort ist ein externes Sortierverfahren, welches nicht in situ arbeitet, jedoch intern implementiert werden kann, wenn ein Array der Länge $2n$ zur Verfügung steht (zum Sortieren von n Daten). Desweiteren kann es adaptiv arbeiten.

- **Auf welcher Idee basiert Merge Sort?**

Merge Sort basiert auf der Idee, daß man zwei sortierte Folgen der Länge n und m in höchstens $m+n-1$ Schritten zu einer sortierten Folge verschmelzen kann.

Dabei vergleicht man jeweils die ersten Elemente einer Folge und schreibt das kleinere in die Ausgabe. Danach geht man auf dem Band, auf dem das kleinere Element stand, ein Element weiter und vergleicht dieses mit dem Element, welches beim letzten Vergleich das größere war. Wenn für eine Folge das Ende erreicht wird, kann der Rest der anderen Folge in die Ausgabe geschrieben werden.

- **Warum genügen höchstens $n+m-1$ Vergleiche?**

Es reichen $n+m-1$ Vergleiche, da mindestens ein Element am Ende ohne Vergleich in die Ausgabe geschrieben wird und nach jedem Vergleich ein Datum in die Ausgabe geschrieben wird.

- **Wie sieht der Algorithmus für Merge Sort aus?**

MERGE SORT $(a_1,...,a_n)$
$(b_1,...,b_{\lceil \frac{n}{2} \rceil}) :=$ MERGESORT $(a_1,...,a_{\lceil \frac{n}{2} \rceil})$
$(c_1,...,c_{\lfloor \frac{n}{2} \rfloor}) :=$ MERGESORT $(a_{\lceil \frac{n}{2}+1 \rceil},...,a_n)$
$(d_1,...,d_n) :=$ MERGE $(b_1,...,b_{\lceil \frac{n}{2} \rceil}; c_1,...,c_{\lfloor \frac{n}{2} \rfloor})$

- **Wieviele Vergleiche braucht Merge Sort im worst case?**

Für $n=2^k$ braucht Merge Sort im worst case $n\log_2(n)-n+1$ Vergleiche. Der worst case besteht darin, daß bei jedem Merge immer nur ein Datum ohne Vergleich in die Ausgabe geschrieben wird. Also:

$$V(n) = V\left(\left\lceil \frac{n}{2} \right\rceil\right) + V\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1$$

Für $n=2^k$ lautet die explizite Lösung:

$$\begin{aligned}
V(2^k) &= V(2^{k-1}) + V(2^{k-1}) + 2^k - 1 \\
&= 2V(2^{k-1}) + 2^k - 1 \\
&= 2\left(2V(2^{k-2}) + 2^{k-1} - 1\right) + 2^k - 1 \\
&= 4V(2^{k-2}) + (2^k - 2) + (2^k - 1) \\
&= 4\left(2V(2^{k-3}) + 2^{k-2} - 1\right) + (2^k - 2) + (2^k - 1) \\
&= 8V(2^{k-3}) + (2^k - 4) + (2^k - 2) + (2^k - 1) \\
&\vdots \\
V(2^k) &= 2^l V(2^{k-l}) + \sum_{i=1}^{l} (2^k - 2^{i-1})
\end{aligned}$$

Wir kennen die spezielle Lösung V(1)=0 und benutzen diese, indem wir $l = k$ setzen:

$$\begin{aligned} V(2^k) &= 2^k V(1) + \sum_{i=1}^{k} \left(2^k - 2^{i-1}\right) \\ &= 0 + \sum_{i=1}^{k} 2^k - \sum_{i=1}^{k} 2^{i-1} \\ &= k \cdot 2^k - (2^k - 1) \text{ gilt wegen } \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \end{aligned}$$

Es gilt $n = 2^k$, also gilt auch $k = \log_2(n)$. Dies setzen wir in die oben gewonnene Formel ein und erhalten:

$$V(n) = n \cdot \log_2(n) - n + 1$$

- **Wie implementiert man Merge Sort auf Bändern?**
  Es gibt zwei Möglichkeiten zur Implementierung auf Bändern:

  1. Zwei-Phasen-Drei-Band-Sortieren
  2. Ein-Phasen-Drei-Band-Sortieren mit Fibonacci-Trick (schneller)

- **Wie funktioniert das Zwei-Phasen-Drei-Band-Sortieren?**
  Nach der j-ten Runde, $0 \leq j \leq \lceil \log_2(n) \rceil$, stehen die Daten auf dem ersten Band. Die Daten an den Positionen $k \cdot 2^j + 1, ..., (k+1)2^j$ für $k \in \mathbb{N}_0$ sollen sortiert sein. Dies ist zu Beginn (nach der 0-ten Runde) sicherlich erfüllt. Dann stehen dort n sortierte Blöcke der Länge 1.

Phase 1: In der Runde j werden die sortierten Blöcke der Länge 2^{j-1} abwechselnd auf das zweite und dritte Band kopiert.

Phase 2: Die zueinander zugehörigen Blöcke, d.h. der m-te Block auf Band 2 und der m-te Block auf Band 3, werden werden zu Blöcken der Länge 2^j auf Band 1 gemischt.

Nach $j = \lceil \log_2(n) \rceil$ ist man sicherlich fertig, da sich die Länge der sortierten Blöcke pro Runde jeweils verdoppelt und man dann einen Block der Länge n hat. Nach Runde 1 Blöcke der Länge $2^1 = 2$. Und nach Runde $\lceil \log_2(n) \rceil$ Blöcke der Länge $2^{\lceil \log_2(n) \rceil} = n$.

- **Wie funktioniert das Ein-Phasen-Drei-Band-Sortieren mit Fibonacci-Trick?**
  Bei diesem Verfahren spart man sich die lästige Kopierarbeit. Wir berechnen in Zeit $O(\log_2(n))$ die kleinste Zahl j mit $Fib(j) \geq n$. Für die Fibonacci-Zahlen gilt:

$$Fib(0) = Fib(1) = 1 \qquad Fib(j) = Fib(j-1) + Fib(j-2), \text{ für } k \geq 2$$

Zu Beginn kopieren wir $Fib(j-1)$ Daten auf Band 2. Die Ausgangsposition sieht dann wie folgt aus:

Band 1: Enthält höchstens $Fib(j-2)$ sortierte Blöcke der Länge $Fib(0) = 1$.

Band 2: Enthält höchstens $Fib(j-1)$ sortierte Blöcke der Länge $Fib(1) = 1$.

Band 3: Ist leer.

Jetzt sortiert man $Fib(j-2)$ zueinander gehörige Blöcke auf Band 3. Nun sieht die Situation wie folgt aus:

Band 1: Ist leer.

Band 2: Enthält höchstens $Fib(j-1) - Fib(j-2) = Fib(j-3)$ sortierte Blöcke der Länge $Fib(1) = 1$.

Band 3: Enthält höchstens $Fib(j-2)$ sortierte Blöcke der Länge $Fib(0) + Fib(1) = Fib(2) = 2$.

Wie man sieht, ist die Ausgangssituation um ein Band verschoben und man kann nach demselben Schema fortfahren, bis man einen sortierten Block der Länge *n* hat.

- **Nach wievielen Runden steht die sortierte Folge auf dem Band?**

Es gilt:

$$Fib(j) = \frac{1}{\sqrt{5}} \cdot \left(\Phi^{j+1} - \hat{\Phi}^{j+1}\right)$$

Es gilt also $Fib(j) \geq n$, wenn gilt:

$$\frac{1}{\sqrt{5}} \cdot \left(\Phi^{j+1} - \hat{\Phi}^{j+1}\right) \geq n$$

Hierbei gilt $\Phi = \left(\frac{1+\sqrt{5}}{2}\right) \approx 1.618$ und $\hat{\Phi} = \left(\frac{1-\sqrt{5}}{2}\right) \approx -0.618$. Wie man sieht ist $\hat{\Phi} \leq 1$ und es gilt $\lim_{j\to\infty} \hat{\Phi}^{j+1} = 0$. Daher kann $\frac{1}{\sqrt{5}}\hat{\Phi}^{j+1}$ mit $\frac{1}{2}$ abgeschätzt werden, da es diesen Wert nie überschreitet. Also:

$$\begin{aligned}
\frac{1}{\sqrt{5}} \cdot \Phi^{j+1} - \frac{1}{2} &\geq n \\
\frac{1}{\sqrt{5}} \cdot \Phi^{j+1} &\geq n + \frac{1}{2} \\
\log_\Phi \frac{1}{\sqrt{5}} + j + 1 &\geq \log_\Phi (n + \frac{1}{2}) \\
j &\geq \log_\Phi n + O(1) \\
j &\geq \log_\Phi 2 \cdot \log_2 n + O(1) \\
j &\geq \frac{\ln 2}{\ln \Phi} \cdot \log_2 n + O(1) \\
j &\geq 1.4404 \cdot \log_2 n + O(1)
\end{aligned}$$

$\log_2(n) \approx \log_2(n + \frac{1}{2})$ gilt wegen:

$$\lim_{n\to\infty} \frac{\log_2(n)}{\log_2(n+\frac{1}{2})} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{n+\frac{1}{2}}} = \lim_{n\to\infty} \frac{n+\frac{1}{2}}{n} = \lim_{n\to\infty} \frac{n}{n} + \lim_{n\to\infty} \frac{\frac{1}{2}}{n} = 1 + 0 = 1$$

Nach ungefähr $1.4404 \cdot \log_2 n$ Runden steht die sortierte Folge also auf einem der Bänder.

- **Wann heißt ein Sortierverfahren adaptiv?**

Ein Sortierverfahren heißt adaptiv, wenn es sich gegenüber vorsortierten Folgen gutmütig verhält.

- **Wie kann man Merge Sort adaptiv machen?**

Ein Lauf ist eine maximale sortierte Teilfolge. Zu Beginn läuft man einmal über das Band und teilt die Folge in ihre Läufe ein. Diese schreibt man abwechselnd auf die zwei Bänder und sortiert diese. Im Falle einer sortierten Eingabe besteht diese aus nur einem Lauf und es muß nichts mehr getan werden.

2.6 Eine untere Schranke für Sortierverfahren

- **Wann heißt ein Sortierverfahren allgemein?**

Ein Sortierverfahren heißt allgemein, wenn es beliebige Folgen $a_1, ..., a_n$ für Daten aus beliebigen (total) geordneten Mengen sortieren kann.

- **Worin besteht das Sortierproblem?**

Das Sortierproblem besteht darin, den Ordnungstyp für eine Eingabe zu berechnen.

- **Was ist ein Ordnungstyp und wieviele Ordnungstypen gibt es für Folgen der Länge n?**

Der Ordnungstyp einer Folge gibt an, an welcher Position ein Datum eigentlich stehen muß, damit die Folge sortiert ist.

So bedeutet der Ordnungstyp $(a_1, a_2, ..., a_i, ..., a_{n-1}, a_n)$, daß das Datum an Position n eigentlich an Position a_n stehen muß. Also entspricht dieser Ordnungstyp folgender Permutation:

$$\pi = \begin{pmatrix} 1 & 2 & \dots & i & \dots & n-1 & n \\ a_1 & a_2 & \dots & a_i & \dots & a_{n-1} & a_n \end{pmatrix}$$

Zum Beispiel bedeutet der Ordnungstyp $(2,1,3)$, daß das erste Datum an die zweite Position muß, das zweite Datum an die erste und das dritte Datum ist schon an der richtigen Position. Folgen, die diesen Ordnungstyp haben sind z.B. $F_1 := 2,1,3; F_2 := 20,10,30; F_3 := 2,-2,99$.

Da die Ordnungstypen Permutationen entsprechen, gibt es also $n!$ verschiedene Ordnungstypen für Folgen der Länge n.

- **Was ist ein Entscheidungsbaum?**

Betrachtet man nur die wesentlichen Vergleiche, die allgemeine Sortierverfahren machen, so lassen sich diese gut in einem Entscheidungsbaum darstellen. Das heißt, an jeder Wurzel des Baumes und seiner Teilbäume wird ein Vergleich gemacht, z.B. $x_1 \leq x_2$?. Gilt $x_1 \leq x_2$ geht man im Entscheidungsbaum links weiter, sonst rechts.

Durch eine Entscheidung wird die Menge der Ordnungstypen in zwei disjunkte Teilmengen zerlegt. Wenn also $x_1 \leq x_2$ gilt, so wird in keinem der folgenden Ordnungstypen gelten, daß x_2 vor x_1 stehen muß.

Ein Sortieralgorithmus stoppt erst, wenn nur noch ein Ordnungstyp möglich ist, daher stehen die Ordnungstypen im Entscheidungsbaum in den Blättern.

- **Wie bestimmt man anhand des Entscheidungsbaumes, wieviele Vergleiche ein allgemeines Sortierverfahren braucht?**

Ist d_π die Tiefe des Weges von der Wurzel, so braucht das Sortierverfahren d_π wesentliche Vergleiche, um den Ordnungstyp π zu berechnen. Allgemein gilt also, daß ein allgemeines Sortierverfahren, so viele Schritte braucht, wie der entsprechende Ordnungstyp tief im Baum ist.

- **Wie bestimmt man eine untere Schranke für den worst case?**

Indem man zeigt, daß jeder binäre Baum mit N Blättern mindestens die Tiefe $\lceil \log_2 N \rceil$ hat. Ein vollständiger binärer Baum weist jeweils bezüglich seiner Tiefe die größte Anzahl an Blättern auf. Ein vollständiger binärer Baum der Tiefe d hat genau 2^d Blätter. Ist die Anzahl der Blätter, die der binäre Baum haben soll, keine Zweierpotenz, d.h. der Form $N = 2^k$, so müssen wir also mindestens bis zur nächsten Zweierpotenz gehen. Für einen Baum der Tiefe d gilt also immer $N \leq 2^d$. Daraus folgt $d \geq \lceil \log_2 N \rceil$.

In unserem Fall ist $N = n!$, da ja $n!$ verschiedene Ordnungstypen in den Blättern des Entscheidungsbaumes stehen. Also hat der Entscheidungsbaum mindestens die

Tiefe $\lceil \log_2 n! \rceil$, d.h es müssen mindestens so viele Vergleiche gemacht werden. Wie wir schon beim Insertion Sort gesehen haben gilt:

$$\log_2(n!) \approx n \log_2 n - 1.4427n$$

Also ist $n \log_2 n - 1.4427n$ eine untere Schranke an Vergleichen für den worst case.

- **Wie bestimmt man eine untere Schranke für den average case?**

Eine untere Schranke für den average case läßt sich bestimmen, indem man zeigt, welche Durchschnittstiefe die Blätter des Entscheidungsbaumes mindestens haben.

Wie berechnet sich die Durchschnittstiefe der Blätter? Der Baum habe N Blätter. Weiter sei anz_d die Anzahl der Blätter auf Tiefe d. Dann ergibt sich die Durchschnittstiefe D der Blätter durch:

$$D = \frac{\sum_{d=0}^{\lceil \log_2 n \rceil} anz_d \cdot d}{N}$$

Angenommen wir haben einen Baum mit Durchschnittstiefe d. Wir können jetzt versuchen, unter Beibehaltung der Anzahl der Blätter die Tiefe zu minimieren. Bei der Minimierung zeigen sich zwei Dinge:

1. Als erstes bemerkt man, daß jedes Blatt auch ein Geschwist haben muß, da man sonst die Durchschnittstiefe senken könnte, indem man das Blatt abschneidet und der Vater als Blatt fungiert.

2. Wenn es also zwei Blätter in Tiefe d gibt, so gibt es keine Blätter in Tiefe $d' \leq d-2$, da man die Durchschnittstiefe sonst wieder senken kann, indem man die zwei Blätter in Tiefe d an das Blatt in der Tiefe d' hängt. Dadurch entsteht ein neues Blatt in Tiefe $d-1$ (der Vater der zwei Blätter) und eines verschwindet auf Tiefe d'. Die anderen beiden Blätter wandern nur von Tiefe d auf Tiefe $d'+1$. Das Verfahren ist also korrekt und die Durchschnittstiefe senkt sich tatsächlich:

$$2(d'+1)+(d-1) = \underbrace{2d'+d+1 \leq d'+2d-1}_{d' \leq d-2} < d'+2d$$

Wie man sieht, ergibt sich folgendes Bild: Ein Baum mit minimaler Durchschnittstiefe hat nur Blätter auf den Ebenen d und $(d-1)$, wenn d die Tiefe des Baumes ist. Desweiteren hat jedes Blatt auf Ebene d ein Geschwist.

Jetzt haben wir schon ein gutes Bild davon, wie ein solcher Baum aussehen muß und wir können konstruktiv (oder eher destruktiv) die Durchschnittstiefe eines Baumes mit N Blättern ermitteln.

Wenn wir N Blätter haben möchten, konstruieren wir erst einmal einen vollständigen binären Baum der Tiefe $\lceil \log_2 N \rceil$. Wie man sieht, haben wir dann entweder genau richtig viele Blätter auf der untersten Ebene, nämlich $2^{\lceil \log_2 N \rceil}$, wenn N eine Zweierpotenz ist, oder wir haben zu viele Blätter.

Da ein Baum mit minimaler Durchschnittstiefe nur Blätter auf den Ebenen d und $(d-1)$ haben kann, können wir überflüssige Blätter streichen. Wir streichen daher jeweils Geschwisterpaare auf Ebene d. Pro Paar, das wir streichen, haben wir ein Blatt weniger. Wie gesagt haben wir $2^{\lceil \log_2 N \rceil}$ Blätter auf Ebene d. N möchten wir am Ende haben, also streichen wir der Differenz entsprechend viele Geschwisterpaare:

$2^{\lceil \log_2 N \rceil} - N$. Auf Ebene $(d-1)$ entstehen entsprechend viele Blätter. Auf Ebene d hatten wir vorher $2^{\lceil \log_2 N \rceil}$ Blätter und streichen $2^{\lceil \log_2 N \rceil} - N$ Geschwisterpaare, also $2 \cdot (2^{\lceil \log_2 N \rceil} - N)$ Blätter:

$$2^{\lceil \log_2 N \rceil} - 2 \cdot (2^{\lceil \log_2 N \rceil} - N) = 2N + 2^{\lceil \log_2 N \rceil} - 2 \cdot 2^{\lceil \log_2 N \rceil} = 2N - 2^{\lceil \log_2 N \rceil}$$

Die Durchschnittstiefe ergibt sich wie folgt:

$$D = \underbrace{\lceil \log_2 N \rceil}_{\text{vollst.bin.Baum}} \underbrace{- \frac{1}{N}(2^{\lceil \log_2 N \rceil} - N)}_{\text{Korrekturterm}} = \lceil \log_2 N \rceil - \frac{2^{\lceil \log_2 N \rceil}}{N} + \frac{N}{N}$$

Wir schätzen dies weiter nach unten ab:

$$D \geq \lceil \log_2 N \rceil - \frac{2^{\lceil \log_2 N \rceil}}{N}$$

Es gilt:

$$1 \leq \frac{2^{\lceil \log_2 N \rceil}}{N} \leq 2$$

Und daher:

$$D \geq \lceil \log_2 N \rceil - 1$$

Auch hier ist $N = n!$, daher ist auch die untere Schranke für die average case-Laufzeit $n \log_2 n - 1.4427n$.

2.7 Bucket Sort

- **Was ist das besondere an Bucket Sort?**

Bucket Sort sortiert n Daten in Zeit $O(n)$, d.h. in linearer Zeit. Daher ist Bucket Sort kein allgemeines Sortierverfahren. Bei Bucket Sort müssen also die Elemente der Menge, aus der diezu sortierenden Daten stammen, bekannt sein. Beziehungsweise das Universum muß bekannt sein.

Man stelle sich eine Wahl vor. Die Parteien, die gewählt werden können, sind bekannt. Daher kann man über die Wahlergebnisse laufen und jeden Stimmzettel in seinen Topf werfen und so die Zettel sortieren.

- **Wie funktioniert Bucket Sort? Welche Laufzeit hat Bucket Sort?**

Angenommen wir haben Daten $a_1,...,a_n \in \{1,...,M\}$, d.h. n Daten aus dem Universum $\{1,...,M\}$.

1. Initialisiere ein Array der Länge M mit M leeren Queues $Q(1),...,Q(M)$.
2. Durchlaufe die Eingabe und hänge a_i an das Ende von $Q(a_i)$ ein, d.h. jedes a_i kommt in seinen Bucket.
3. Hänge $Q(1),...,Q(M)$ aneinander.

Die Laufzeit beträgt $O(n+M)$. Es werden n Daten betrachtet und M Queues initialisiert bzw. aneinander gehängt.

- **Wie ist die lexikographische Ordnung definiert?**

$$(b_k,...,b_1) < (c_k,...,c_1) \Leftrightarrow (\exists b_i < c_i) \wedge (b_j = c_j; \forall j > i)$$

- **Wie kann man mit Bucket Sort lexikographisch sortieren? Welche Laufzeit hat dieses Verfahren?**

Angenommen wir haben n Daten $a_1,...,a_n \in \{1,...,M\}^l$ versehen mit der lexikographischen Ordnung $\square < A < B < ... < Z$ (mit $\square$=Leerzeichen), d.h. $a_i = (a_l,...,a_1)$.

1. Führe l Runden des Bucket Sort-Algorithmus aus, wobei in der k-ten Runde die Wörter a_i als ganzes bezüglich ihrer k-ten Buchstaben a_{ik} einsortiert werden.

Die Laufzeit beträgt $O(l(n+M))$. l-mal wird der Bucket Sort-Algorithmus mit Laufzeit $O(n+M)$ ausgeführt.

Dieser Algorithmus arbeitet auch korrekt. Sei $a_i < a_j$ mit $a_{ik} < a_{jk}$ und $a_{im} = a_{jm}$ für $m > k$. Dann wird in Runde k das Wort a_i vor das Wort a_j soertiert. Dies bleibt auch so, da wir Queues benutzen.

- **Wie kann man den Algorithmus zum lexikographischen Sortieren noch optimieren?**

Wir haben noch nicht beachtet, daß nicht alle Wörter immer gleich lang sind, daß also Wörter unter Umständen viele Leerzeichen am Ende haben.

Sei l_i die Länge des i-ten Wortes, l_{max} die Länge des längsten Wortes und l_{total} die Summe aller l_i.

Zuerst machen wir einen Bucket Sort bezüglich der Wortlänge und fügen die Wörter in Listen LIST(l_i) ein, wobei LIST(l_i) die Wörter der Länge l_i enthält. Danach werden l Runden des oben beschriebenen Algorithmus zum lexikographischen Sortieren durchgeführt. In der ersten Runde benutzen wir nur die Wörter aus LIST(l_{max}). In

der k-ten Runde hängen wir die Wörter aus LIST$(l_{max} - k + 1)$ vor das Ergebnis und sortieren dann. Dieses Vorgehen ist korrekt, da diese Wörter an den hinteren Positionen nur Leerzeichen haben.

Die Laufzeit beträgt $O(l_{total} + l_{max}M)$. Denn jeder Buchstabe jedes Wortes wird mal einsortiert (dies sind l_{total}) und l_{max}-mal werden die Queues initialisiert und aneinander gehängt.

2.8 Batcher Merge

- **Wie funktioniert Batcher Sort?**

Batcher Sort funktioniert ähnlich wie der normale Merge Sort, nur werden hier die rekursiven Batcher Sort-Aufrufe parallel behandelt. Als Eingabe bekommt der Algorithmus irgendeine Folge $a_1...a_n$:

1. $n = 1$: STOP
2. $n \geq 2$:
   $(b_1...b_{\frac{n}{2}}) = \text{BATCHERSORT}(a_1...a_{\frac{n}{2}})$
   $(c_1...c_{\frac{n}{2}}) = \text{BATCHERSORT}(a_{\frac{n}{2}+1}...a_n)$
3. $(d_1...d_n) = \text{BATCHERMERGE}(b_1...b_{\frac{n}{2}}c_1...c_{\frac{n}{2}})$

- **Wie funktioniert Batcher Merge?**

Die Eingabe seien zwei sortierte Folgen $a_1...a_n$ und $b_1...b_n$:

1. $n = 1 \Rightarrow z_1 = \min\{a_1, b_1\}, z_2 = \max\{a_1, b_1\}$
2. $n \geq 2$:
   $(v_1...v_n) = \text{BM}(a_1, a_3, ..., a_{n-1}, b_1, b_3, ..., b_{n-1})$
   $(w_1...w_n) = \text{BM}(a_2, a_4, ..., a_n, b_2, b_4, ..., b_n)$
   Diese beiden Aufrufe werden parallel verarbeitet.
3. Vergleiche parallel v_{i+1} und w_i für $1 \leq i \leq n-1$.
   Die Ergebnisfolge z ergibt sich aus:

$$\begin{aligned} z_1 &= v_1 \\ z_{2i} &= \min\{v_{i+1}, w_i\} \\ z_{2i+1} &= \max\{v_{i+1}, w_i\} \\ z_{2n} &= w_n \end{aligned}$$

- **Zeige, daß Batcher Merge ein korrekter Mischalgorithmus ist!**

Zuerst kann man beobachten, daß ein a_j gilt in der sortierten Folge z die potentiellen Positionen j (alle b sind größer) bis $j+n$ (alle b sind kleiner) hat.

- **Welche Laufzeit hat Batcher Merge?**

Batcher Merge mischt zwei Folgen der Länge $n = 2^k$ mit $n\log_2 n + 1$ Vergleichen in paralleler Zeit $\log_2 n + 1$. Denn es gilt:

1. Sei $M(n)$ die Anzahl an Vergleichen, die für eine Folge der Länge n gemacht werden. Dann wird Batcher Merge zweimal für Folgen der halben Länge aufgerufen und jede Folge verursacht $n-1$ Vergleiche.

$$M(n) = 2M\left(\frac{n}{2}\right) + n - 1$$

Für $n = 2^k$:

$$\begin{aligned} M(2^k) &= 2M(2^{k-1}) + 2^k - 1 \\ &= 2(2M(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \\ &= 4M(2^{k-2}) + (2^k - 2) + (2^k - 1) \\ &= 8M(2^{k-3}) + (2^k - 4) + (2^k - 2) + (2^k - 1) \\ &= 2^l M(2^{k-l}) + \sum_{i=1}^{l} (2^k - 2^{i-1}) \end{aligned}$$

Wir benutzen die spezielle Lösung $l = k$, da wir wissen, daß $M(1) = 1$ ist.

$$\begin{aligned} M(2^k) &= 2^k M(1) + \sum_{i=1}^{k} (2^k - 2^{i-1}) \\ &= 2^k + k \cdot 2^k - 2^k - 1 \\ &= k \cdot 2^k - 1 \end{aligned}$$

Also gilt wegen $n = 2^k$ und $k = \log_2 n$:

$$M(n) = n \log_2 n - 1$$

2. Sei $PM(n)$ die parallele Zeit fürs Mischen, dann werden zwei Folgen der halben Länge parallel gemischt.

$$PM(n) = PM\left(\frac{n}{2}\right) + 1$$

Und für $n = 2^k$:

$$\begin{aligned} PM(2^k) &= PM(2^{k-1}) + 1 \\ &= (PM(2^{k-2}) + 1) + 1 \\ &= PM(2^{k-3}) + 1 + 1 + 1 \\ &= PM(2^{k-l}) + \sum_{i=1}^{l} 1 \end{aligned}$$

Auch hier benutzen wir die spezielle Lösung $l = k$, da wir $PM(1) = 1$ kennen.

$$\begin{aligned} PM(n) &= PM(1) + \sum_{i=1}^{k} 1 \\ &= 1 + k \end{aligned}$$

Wieder gilt $k = \log_2 n$, also gilt für n:

$$PM(n) = \log_2 n + 1$$

2.9 Das Auswahlproblem

- **Was ist das Auswahlproblem? Worin besteht das Medianproblem?**

Das Auswahlproblem besteht darin, aus n Daten $a_1, ..., a_n$ mit Rängen $k \in \{1, ..., n\}$ das Datum mit Rang k zu bestimmen. Man spricht vom Medianproblem für $k = \lceil \frac{n}{2} \rceil$.

- **Gebe einen Algorithmus an, mit dem sich das Auswahlproblem lösen läßt!**

Gegeben sei eine Eingabe $a_1, ..., a_n$. Gesucht wird das Datum mit Rang k.

1. Wähle zufällig ein $i \in \{1, ..., n\}$ aus und bestimmt den Rang r des Elementes a_i durch Vergleich mit allen anderen. Dabei werden die Elemente in die drei Klassen SMALL, LARGE und EQUAL eingeteilt. Die Klasse SMALL enthält alle Daten, die kleiner sind als a_i, in Klasse LARGE sind alle größeren Daten und die Klasse EQUAL enthält a_i.

2. Falls r=k, ist a_i das gesuchte Datum.

   Falls $r > k$, suche in SMALL nach dem Datum mit Rang k.

   Falls $r < k$, suche in LARGE nach dem Datum mit Rang $k - r$.

Es ist offensichtlich, daß dieser Algorithmus korrekt arbeitet.

- **Wie ist die Laufzeit des Algorithmus?**

Es entstehen mit jeweils Wahrscheinlichkeit $\frac{1}{n}$ Probleme der Größe:

$$\underbrace{\overbrace{n-1}^{r=1}, ..., \overbrace{n-(k-1)}^{r=(k-1)}}_{\sum_{i=n-(k-1)}^{n-1} V(i)}, \overbrace{0}^{r=k}, \underbrace{\overbrace{k}^{r=(k+1)}, ..., \overbrace{n-1}^{r=n}}_{\sum_{i=k}^{n-1} V(i)}$$

Wir schätzen $V(n)$ nach oben ab durch:

$$V(n) \leq n - 1 + \frac{1}{n} \left(\sum_{i=k}^{n-1} V(i) + \sum_{i=n-(k-1)}^{n-1} V(i) \right)$$

$V(n)$ ist eine monoton wachsende Funktion, daher wird $(\sum + \sum)$ maximal für $k = \lceil \frac{n}{2} \rceil$. Daher gilt weiterhin:

$$V(n) \leq n - 1 + \frac{1}{n} \left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} V(i) + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} V(i) \right)$$

Die durchschnittliche Problemgröße für $\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} V(i)$ ist $\frac{3}{4}n$. Da $1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + ... = 4$ vermuten wir $V(n) \leq 4n$. Dies weisen wir jetzt mit einem Induktionsbeweis nach.

$$V(n) \quad \leq \quad n - 1 + \frac{1}{n} \left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} 4i + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} 4i \right)$$

2.10 Zusammenfassung

Problem	Laufzeit
Insertion Sort VG	$n\log_2 n - 0.4427n + O(\log_2 n)$
Insertion Sort OP	$\Theta(n^2)$
Quick Sort WC	$\Theta(n^2)$
Quick Sort AC Fall 1 und 2	$2(n+1)H(n) - 4n =$ $1.386n\log_2 n - 2.846n + 1.386\log_2 n + 1.154$
Quick Sort AC Fall 3 und 4	$\frac{12}{7}(n+1)H(n-1) - \frac{477}{147}n + \frac{223}{147} + \frac{252}{147n} =$ $1.188n\log_2 n - 2.255n + 1.188\log_2(n-1) + 2.507$
Heap Sort ST_{wc}	$2n\log_2 n + 2n$
Heap Sort ST_{ac}	$2n\log_2 n - O(n)$
Heap Sort BU_{bin} WC	$n\log_2 n + n\log_2^2 n + 3n$
Heap Sort BU_{lin} AC	$n\log_2 n + O(n)$
Heap Sort BU_{lin} WC	$1.5n\log_2 n + O(n)$
Merge Sort AC,WC	$n\log_2 n - n + 1$

3 Dynamische Dateien

3.1 Vorbemerkung

Dynamische Dateien sind zum Speichern und zum Löschen von Daten da. Im folgenden wird davon ausgegangen, daß ein Datum aus einem Schlüssel $k \in U$ und dem eigentlichen Datensatz besteht. Dabei ist U ein (geordnetes oder ungeordnetes) Universum.

- **Welche Operationen sollten dynamische Dateien unterstützen?**

  1. MAKEDICT: Erzeuge eine leere Datei.
  2. SEARCH(k): Suche das Datum, welches unter dem Schlüssel k abgespeichert ist.
  3. INSERT(k,x): Füge das Datum x unter dem Schlüssel k ein. Wenn es den Schlüssel k schon gibt, wird das Datum überschrieben.
  4. DELETE(k): Lösche den unter k abgespeicherten Datensatz.

- **Worin unterscheiden sich Hashing und binäre Suchbäume?**
  Hashing hat eine schlechte worst-case-Laufzeit, unter Umständen jedoch ein gutes average-case-Verhalten. Mit Suchbäume und balancierten Suchbäumen wird das worst-case-Verhalten kontrolliert. Jedoch muß das Universum hier eine vollständig geordnete Menge sein.
- **Welche Operationen werden von einigen Datenstrukturen zusätzlich unterstützt?**

  1. MIN (MAX): Finde das unter dem kleinsten (größten) Schlüssel gespeicherte Datum.
  2. LIST: Liste die Daten nach ihren Schlüsseln sortiert auf.
  3. CONCATENATE(D_1, D_2, D): Hänge die Daten aus D_1 und D_2 in eine neue Datei D aneinander. Geht nur, wenn alle Daten aus D_1 kleiner als alle Daten in D_2 sind.
  4. SPLIT(D, k, D_1, D_2): Teile die Datei D in zwei Dateien D_1 und D_2, wobei D_1 alle Datensätze k' mit $k' < k$ enthält und D_2 alle Datensätze, deren Schlüssel größer als k sind.

3.2 Hashing

Wir nehmen an, daß das Universum U bekannt ist und daß wir tatsächlich viel weniger als $|U|$ Daten speichern möchten. Dabei werden die Daten in einem Array der Länge M mit den Adressen $0,...,M-1$ gespeichert.

- **Was ist eine Hashfunktion?**

Eine Hashfunktion bildet das Universum auf die Adressen des Arrays ab,d.h.:

$$h : U \longrightarrow \{0,...,M-1\}$$

- **Welche Eigenschaften sollte eine gute Hashfunktion haben?**

1. Sie sollte schnell auszuwerten sein. Nach Möglichkeit in $O(1)$.
2. Sie sollte gut streuen, d.h. die Daten gleichmäßig verteilen. Optimal: Gleichverteilung $\frac{1}{M}$.

- **Warum kann eine Hashfunktion nicht injektiv sein?**

Wegen $|M| \ll |U|$.

- **Was ist eine Kollision?**

Bei einer Kollision sollen zwei verschiedene Daten an derselben Adresse gespeichert werden, d.h.:

$$x \neq x' \wedge h(x) = h(x')$$

Kollisionen sind schon bei ideal streuenden Hashfunktionen mit weit weniger als M Daten wahrscheinlich.

- **Zeige, daß man mit Kollisionen leben muß!**

Dies kann man mit dem Geburtstagsparadoxon zeigen. Das Geburtstagsparadoxon besagt, daß schon 23 Leute reichen, damit die Wahrscheinlichkeit, daß zwei Leute am selben Tag Geburtstag haben ca. 50 % beträgt.

Also ist hier $n = 23$ und $M = 365$. Die Wahrscheinlichkeit, daß j an einer bestimmten Adresse abgespeichert wird, beträgt für jede Adresse $\frac{1}{M}$. Die Gesamtwahrscheinlichkeit ergibt sich durch:

$$\frac{M}{M} \cdot \frac{M-1}{M} \cdot \frac{M-2}{M} \cdot \ldots \cdot \frac{M-n+1}{M}$$

Für $n = 50$ beträgt die Wahrscheinlichkeit sogar nur noch 0.03%. Bei 50 abzuspeichernden Daten und insgesamt 365 Speicherplätzen, muß man also schon stark mit Kollisionen rechnen.

- **Welche Funktion hat sich als Hashfunktion bewährt?**

Für Universen $U = \{0,...,u-1\}$ mit $u \in \mathbb{N}$ hat sich die Modulo-Funktion bewährt:

$$h(x) \equiv x \mod M$$

Dabei sollte M nach Möglichkeit prim sein, da nur die ersten Stellen weggestrichen werden, wenn M eine Zweierpotenz ist, d.h. $M = 2^k$ mit $k \in \mathbb{N}_0$. Dies kann bei strukturierten Daten zu übermäßig vielen Kollisionen führen.

- **Welche Methoden zur Kollisionsbehandlung gibt es?**

1. Lineares Sondieren (Linear Probing)

2. Quadratischen Sondieren
3. Add To Hash
4. Doppeltes Hashing
5. Offenes Hashing

- **Wie funktioniert das lineare Sondieren?**

Bei einer Kollision werden alle Plätze nach der Stelle, wo x eigentlich gespeichert werden sollte überprüft, bis ein freier Platz gefunden wurde.

Für jeden Speicherplatz werden zwei zusätzliche boolesche Variablen *empty* und *deleted* verwaltet. Bei $empty = 1$ ist der Platz frei. $deleted = 0$ bedeutet, daß noch kein Element an diesem Speicherplatz gelöscht wurde, während $deleted = 1$ das Gegenteil bedeutet.

- **Wie funtionieren INSERT, SEARCH und DELETE beim linearen Sondieren?**

1. INSERT(x): Wir gehen davon aus, daß x noch nicht gespeichert wurde. Sonst müßten wir x erst suchen. Berechne $h(x)$ und suche das kleinste $i \in \{0,...,M-1\}$, so daß $(h(x)+i) \mod M$ ein freier Platz ist. Sollen mehr als M Daten gespeichert werden, kommt es zu einem Overflow, da dann das komplette Array einmal durchsucht wurde.

2. SEARCH(x): Berechne $h(x)$ und durchlaufe die Speicherplätze $(h(x)+i) \mod M$, bis x gefunden wurde, oder ein Speicherplatz mit $emtpy = 1$ und $deleted = 0$. Nur dann können wir sicher sein, daß x nicht noch später folgt.

3. DELETE(x): Wir suchen x mit SEARCH(x) und setzen $empty = 1$ und $deleted = 1$.

- **Was ist der Nachteil beim linearen Sondieren?**

Bei einer relativ konstanten Zahl an Daten, die sich jedoch häufig ändern, gibt es relativ schnell kaum noch Plätze mit $deleted = 0$, so daß eine Suche nach einem nicht vorhandenen Element stets alle Speicherplätze besucht. Zusätzlich tendiert das Lineare Sondieren zur Klumpenbildung.

- **Warum tendiert das Lineare Sondieren zur Klumpenbildung?**

Weil Elemente oft am Ende von langen Ketten eingefügt werden.

Beispiel:

$Prob(h(x) = i)$ sei für $0 \leq i \leq 7$ die Wahrscheinlichkeit, daß x an Position i eingefügt werden soll. Es gelte $Prob(h(x) = i) = \frac{1}{8}$ für $0 \leq i \leq 7$, d.h. alle Plätze sind gleich wahrscheinlich. Die Speicherplätze $0,3,4,5$ seien belegt. Dann ist die Wahrscheinlichkeit für die Belegung der leeren Plätze:

Position 1: $Prob(h(x) = 1) = \frac{1}{4}, h(x) \in \{0,1\}$
Position 2: $Prob(h(x) = 2) = \frac{1}{8}, h(x) \in \{2\}$
Position 6: $Prob(h(x) = 6) = \frac{1}{2}, h(x) \in \{3,4,5,6\}$
Position 7: $Prob(h(x) = 7) = \frac{1}{8}, h(x) \in \{7\}$

Also tendiert auch eine Suche dazu lange zu dauern. Knuth hat folgende Ergebnisse für den average case bewiesen, wenn es nur INSERT- und SEARCH-Befehle gibt:

Auslastung	erfoglose Suche/INSERT	erfolgreiche Suche
20%	1.28	1.125
50%	2.5	1.5
80%	13	3
90%	50.5	5.5
95%	200.5	10.5

- **Wie funktioniert das Quadratische Sondieren?**

Für x werden folgende Speicherplätze ausprobiert:

$$\begin{aligned} h(x) &\mod M \\ (h(x)+1^2) &\mod M \\ (h(x)-1^2) &\mod M \\ (h(x)+2^2) &\mod M \\ &\vdots \\ (h(x)+i^2) &\mod M \\ (h(x)-i^2) &\mod M \end{aligned}$$

Wenn M eine Primzahl ist und $M \equiv 3 \mod 4$, bzw. M läßt sich schreiben als $M = 4 \cdot i + 3$, dann werden alle Speicherplätze getroffen (Ergebnis aus der Zahlentheorie).

Das Problem der Primärkollision ($h(x) = h(y)$) bleibt unberührt, wenn jedoch x an der Stelle $(h(x)+i^2) \mod M$ gespeichert wurde und $h(y) = (h(x)+i^2)$, d.h. y soll primär an der Stelle abgespeichert werden, an der x nach einer Suche gespeichert wurde, und weiter gelte $h(z) = h(x)$, d.h. auch z soll an der Stelle gespeichert werden, wo primär x gespeichert werden sollte, dann werden bei der Suche nach freien Plätzen verschiedene Folgen durchlaufen.

- **Wie funktioniert das Verfahren Add To Hash?**

Beim verfahren Add To Hash, welches ähnliche Eigeschaften, wie das quadratische Sondieren hat, werden folgende Speicherplätze ausprobiert:

$$\begin{aligned} h(x) &\mod M \\ 2h(x) &\mod M \\ 3h(x) &\mod M \\ 4h(x) &\mod M \\ &\vdots \end{aligned}$$

- **Zeige, daß beim Verfahren Add To Hash alle Speicherplätze während der ersten M Versuche getroffen werden!**

Die Behauptung sieht wie folgt aus: Wenn M eine Primzahl ist, $j \in \{1,...,M-1\}$ und $g(i) \equiv ij \mod M$, so sind $g(0), g(1), ..., g(M-1)$ paarweise verschieden.

Beweis:

Angenommen es existieren i, i' mit $g(i) = g(i') \wedge 0 \leq i < i' \leq M-1$. Dann gilt also: $ij \equiv i'j \mod M$, d.h. ij und $i'j$ lassen bei der Division durch die Zahl M denselben Rest. Daher gilt also auch:

$$i'j \equiv ij \mod M \Leftrightarrow i'j - ij \equiv 0 \mod M \Leftrightarrow (i'-i)j \equiv 0 \mod M$$

$(i'-i)\,j$ ist also ein Vielfaches der Primzahl M. $j \in \{1,...,M-1\}$, also ggT$(j,M)=1$, da $j < M$ ist und M eine Primzahl. Jedoch ist auch $(i'-i) \in \{1,...,M-1\}$, also ist auch der ggT$((i'-i),M)=1$, was jedoch ein Widerspruch ist.

- **Wie funktioniert das doppelte Hashing?**

Beim doppelten Hashing benutzt man zwei unabhängige Hashfunktionen $h(x) = x \mod p$ und $h'(x) = x \mod q$ mit p und q prim und $p \neq q$. Es werden jetzt für ein x folgende Speicherplätze ausprobiert:

$$\begin{array}{ll} \left(h(x)+h'(x)\cdot 1^2\right) & \mod M \\ \left(h(x)+h'(x)\cdot 2^2\right) & \mod M \\ \left(h(x)+h'(x)\cdot 3^2\right) & \mod M \\ \vdots & \end{array}$$

Es hat sich gezeigt, daß das doppelte Hashing dem Verhalten des idealen Hashing sehr nahe kommt.

- **Wie funktioniert das offene Hashing?**

Beim offenen Hashing wird an jeden Speicherplatz eine lineare Liste angelegt. Kollisionsbehandlung ist hier nicht nötig, da jedes x an seinem Platz $h(x)$ abgelegt wird. Es wird nämlich einfach an den Anfang der linearen Liste platziert. Dies geht in $O(1)$.

Jedoch kann man ein Element jetzt nicht mehr in konstanter Zeit finden. Da man die lineare Liste durchlaufen muß.

Löschen ist auch in $O(1)$ möglich, wenn man sich bei der vorherigen erfolgreichen Suche den Zeiger auf x merkt.

- **Was ist ideales Hashing?**

Beim idealen Hashing wird angenommen, daß alle Konfigurationen von n besetzten Speicherplätzen aus einer Gesamtzahl von M Speicherplätzen gleich wahrscheinlich sind. Sind n Speicherplätze besetzt so gibt es insgesamt $\binom{M}{n}$ verschiedene Konfigurationen.

$$P_r = \frac{\binom{M-r}{n-(r-1)}}{\binom{M}{n}}$$

Desweiteren gilt:

$$P_r = 0 \text{ für } r > n+1$$

Der Erwartungswert ergibt sich aus der Zufallsvariablen multipliziert mit der Wahrscheinlichkeit der Zufallsvariablen. Also:

$$\begin{aligned} E &= \sum_{r=1}^{n+1} r \cdot P_r \\ &= \sum_{r=1}^{n+1} r \cdot \frac{\binom{M-r}{n-(r-1)}}{\binom{M}{n}} \end{aligned}$$

Unter Ausnutzung der Symmetrie des Binomialkoeffizienten ($\binom{a}{b} = \binom{a}{a-b}$) gilt weiter:

$$\begin{aligned} E &= \sum_{r=1}^{n+1} r \cdot \frac{\binom{M-r}{M-r-n+r-1}}{\binom{M}{n}} \\ &= \sum_{r=1}^{n+1} r \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} \end{aligned}$$

Jetzt wird r durch $(M+1)-(M+1-r)$ ersetzt, was legitim ist, da die Formel $r = (M+1)-(M+1-r)$ eine wahre Aussage ist. Also:

$$\begin{aligned} E &= \sum_{r=1}^{n+1} ((M+1)-(M+1-r)) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} \\ &= \sum_{r=1}^{n+1} \left((M+1) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} - (M+1-r) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} \right) \\ &= \sum_{r=1}^{n+1} (M+1) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} - \sum_{r=1}^{n+1} (M+1-r) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} \\ &= (M+1) \cdot \sum_{r=1}^{n+1} \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} - \sum_{r=1}^{n+1} (M+1-r) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} \end{aligned}$$

Da $\sum_{r=1}^{n+1} \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}} = \sum_{r=1}^{n+1} P_r = 1$ ist, gilt:

$$E = M+1 - \sum_{r=1}^{n+1} (M+1-r) \cdot \frac{\binom{M-r}{M-n-1}}{\binom{M}{n}}$$

Aus irgendwelchen obskuren mathematischen Gründen gilt:

$$(M+1-r) \cdot \binom{M-r}{M-n-1} = (M-n) \cdot \binom{M-r+1}{M-n}$$

$$\begin{aligned} E &= M+1 - \sum_{r=1}^{n+1} (M-n) \cdot \frac{\binom{M-r+1}{M-n}}{\binom{M}{n}} \\ &= M+1 - \frac{(M-n)}{\binom{M}{n}} \cdot \sum_{r=1}^{n+1} \binom{M-r+1}{M-n} \end{aligned}$$

Aus irgendeinem weiteren noch obskureren Grund gilt auch:

$$\sum_{r=1}^{n+1} \binom{M-r+1}{M-n} = \binom{M+1}{M-n+1}$$

Insgesamt ergibt dies also:

$$\begin{aligned} E &= M+1 - (M-n) \cdot \frac{\binom{M+1}{M-n+1}}{\binom{M}{n}} \\ &= M+1 - (M-n) \cdot \frac{\binom{M+1}{n}}{\binom{M}{n}} \end{aligned}$$

Betrachten wir $\frac{\binom{M+1}{n}}{\binom{M}{n}}$:

$$\frac{\binom{M+1}{n}}{\binom{M}{n}} = \frac{(M+1)! \cdot n! \cdot (M-n)!}{n! \cdot (M+1-n)! \cdot M!} = \frac{M+1}{M-n+1}$$

$$\begin{aligned} E &= M+1-(M-n)\cdot\frac{M+1}{M-n+1} \\ &= (M+1)\left(1-\frac{M-n}{M-n+1}\right) \\ &= (M+1)\cdot\left(\frac{M-n+1}{M-n+1}-\frac{M-n}{M-n+1}\right) \\ &= (M+1)\cdot\frac{1}{M-n+1} \\ &= \frac{M+1}{M-n+1} \end{aligned}$$

Dies bedeutet, daß man bei n gespeicherten Daten mit einem Gesamtspeicher von M erwartungsgemäß $\frac{M+1}{M-n+1}$ Plätze testen muß, bis ein freier Platz gefunden wird. Einen Wert bezüglich des Auslastungsfaktor bekommt man, indem man für n zum Beispiel $\frac{M}{2}$ einsetzt. Dann wäre die Auslastung bei 50%:

$$\begin{aligned} E &= \frac{M+1}{M-\frac{M}{2}+1} \\ &= \frac{M+1}{\frac{M}{2}+1} \\ &= \frac{M\cdot(1+\frac{1}{M})}{M\cdot(\frac{1}{2}+\frac{1}{M})} \\ &= \frac{1+\frac{1}{M}}{\frac{1}{2}+\frac{1}{M}} \end{aligned}$$

Desweiteren gilt:

$$\lim_{M\to\infty}\frac{1+\frac{1}{M}}{\frac{1}{2}+\frac{1}{M}} = \frac{\lim_{M\to\infty}(1+\frac{1}{M})}{\lim_{M\to\infty}(\frac{1}{2}+\frac{1}{M})} = \frac{\lim_{M\to\infty}1+\lim_{M\to\infty}\frac{1}{M}}{\lim_{M\to\infty}\frac{1}{2}+\lim_{M\to\infty}\frac{1}{M}} = \frac{1+0}{\frac{1}{2}+0} = 2$$

Bei einer Auslastung von 50% muß ich also erwartungsgemäß 2 Plätze testen.

- **Wieviele Plätze werden bei einer erfolgreichen Suche mit idealem Hashing getestet?**

Wenn das Datum gesucht wird, welche als k-tes Datum eingefügt wurde, so sind die Kosten dafür genau so hoch wie beim Einfügen dieses Datums. Die erwartete Anzahl an Tests liegt also für das k-te Datum bei $\frac{M+1}{M-(k-1)+1} = \frac{M+1}{M-k+2}$. Die n Daten werden mit Wahrscheinlichkeit $\frac{1}{n}$ gesucht, daraus ergibt sich:

$$\begin{aligned} E_{such} &= \frac{1}{n}\cdot\sum_{k=1}^{n}\frac{M+1}{M-k+2} \\ &= \frac{M+1}{n}\cdot\sum_{k=1}^{n}\frac{1}{M-k+2} \\ &= \frac{M+1}{n}\cdot\left(\frac{1}{M-n+2}+\frac{1}{M-n+3}+\ldots+\frac{1}{M+1}\right) \end{aligned}$$

Es sei hierbei angemerkt, daß in der Klammer die Summe von n in Richtung 1 aufgelöst wurde und daß $\frac{1}{M+1}$ das größte Glied ist. In der Klammer steht also die Harmonische Reihe bis $\frac{1}{M+1}$ nur daß sie bei $\frac{1}{M-n+2}$ startet. Es fehlt also die Reihe bis $\frac{1}{M-n+1}$. Also gilt:

$$\begin{aligned} E_{such} &= \frac{M+1}{n} \cdot (H(M+1) - H(M-n+1)) \\ &= \frac{M+1}{n} \cdot (\ln(M+1) + \gamma - \ln(M-n+1) - \gamma) \\ &= \frac{M+1}{n} \cdot (\ln(M+1) - \ln(M-n+1)) \\ &= \frac{M+1}{n} \cdot \ln \frac{M+1}{M-n+1} \end{aligned}$$

3.3 Binäre Suchbäume

Beim Hashing muß das Universum keine geordnete Struktur haben. Bei binären Suchbäumen muß das Universum jedoch eine geordnete Menge sein.

- **Wann heißt ein binärer Baum Suchbaum?**

Ein binärer Baum heißt Suchbaum, wenn für jeden Knoten v mit Datum x gilt, daß im linken Teilbaum nur Daten $x' < x$ enthalten sind und im rechten Teilbaum nur Daten $x'' > x$.

- **Was wird in den inneren Knoten gespeichert, wenn alle Daten in den Blättern stehen müssen?**

Die inneren Knoten enthalten dann ,,Routing"-Informationen, d.h. das größte im linken Teilbaum gespeicherte Datum. (Siehe 2-3-Bäume)

- **Wie funktioniert SEARCH(x)?**

SEARCH(x): Wir starten an der Wurzel und vergleichen x mit dem enthaltenen Datum y. Es gibt drei Fälle:

1. $x = y$. Suche erfolgreich beenden.
2. $x < y$. Gehe in den linken Teilbaum und suche dort weiter.
3. $x > y$. Gehe in den rechten Teilbaum und suche dort weiter.

- **Wie funktioniert INSERT(x)?**

INSERT(x): Nach erfolgloser Suche. Der bei der erfolglosen Suche gefundene *NIL*-Zeiger zeigt auf einen neuen Knoten mit x als Datum. Weiter enthält dieser Knoten zwei *NIL*-Zeiger.

- **Wie funktioniert DELETE(x)?**

Nach einer erfolgreichen Suche haben wir den Knoten v, der x enthält, gefunden. Wir unterscheiden drei Fälle:

1. v ist ein Blatt. Setze den Zeiger auf v auf *NIL*.
2. v hat ein Kind. Setze den Zeiger auf v auf das Kind von v.
3. v hat zwei Kinder. Suche im Suchbaum den Inorder-Vorgänger y von x und vertausche die beiden. Jetzt gilt für x entweder (1) oder (2). Der Inorder-Vorgänger wird wie folgt gefunden: Gehe von x aus einen nach links und dann immer rechts, bis ein *NIL*-Zeiger kommt. Der Knoten, zu dem dieser *NIL*-Zeiger gehört, enthält das gesuchte Datum y.

- **Wie groß ist die average-case-Suchzeit in einem binären Suchbaum mit n Daten? Beziehungsweise: Wie tief ist ein zufällig gebildeter Suchbaum über n Daten durchschnittlich?**

$$R(n) = 1 + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{i-1}{n}R(i-1) + \frac{n-i}{n}R(n-i)\right)$$

Wie ergibt sich diese Formel? Um diese Frage zu beantworten, muß man sich überlegen, wie ein durchschnittlicher binärer Suchbaum aussieht, bzw. wie tief dieser ist.

Die 1 ergibt sich dadurch, daß man mindestens einen Vergleich an der Wurzel durchführen muß, bzw. dadurch, daß die Wurzel mindestens die Höhe 1 verursacht. Der Faktor $\frac{1}{n}$ vor der Summe ergibt sich dadurch, daß jedes Element gleichwahrscheinlich an der Wurzel steht. Wenn das i-kleinste Element an der Wurzel steht, stehen links davon $(i-1)$ Elemente mit einer Wahrscheinlichkeit von $\frac{i-1}{n}$ und rechts davon $(n-i)$ Elemente mit einer Wahrscheinlichkeit von $\frac{n-i}{n}$. Wir erhalten also die obige Rekursionsgleichung:

$$\begin{aligned} R(n) &= 1+\frac{1}{n}\sum_{i=1}^{n}\left(\frac{i-1}{n}R(i-1)+\frac{n-i}{n}R(n-i)\right) \\ n\cdot R(n) &= n+\sum_{i=1}^{n}\left(\frac{i-1}{n}R(i-1)+\frac{n-i}{n}R(n-i)\right) \\ n\cdot R(n) &= n+\frac{1}{n}\sum_{i=1}^{n}\left((i-1)R(i-1)+(n-i)R(n-i)\right) \end{aligned}$$

Wir setzen $T(n) = n\cdot R(n)$, also $R(n) = \frac{T(n)}{n}$.

$$T(n) = n+\frac{1}{n}\sum_{i=1}^{n}\left(T(i-1)+T(n-i)\right)$$

Dies ist fast die gleiche Formel, wie bei der Quicksort-Analyse:

1. QS: $V_{ac}(n) = n-1+\frac{1}{n}\sum_{i=1}^{n}\left(V_{ac}(i-1)+V_{ac}(n-i)\right)$ mit $V_{ac}(1)=0$
2. BS: $T(n) = n+\frac{1}{n}\sum_{i=1}^{n}\left(T(i-1)+T(n-i)\right)$ mit $T(1)=1$

Der Unterschied besteht also in den unterschiedlichen Summanden $n-1$ und n. Die Differenz bei der Auflösung von $T(n)$ beträgt weniger als n. Desweiteren gilt $T(1)=1$, im Gegensatz zu $V_{ac}(1)=0$ bei Quicksort. Auch dies ergibt bei Auflösung höchstens eine Differenz von n. Wir können $T(n)$ also wie folgt abschätzen:

$$\begin{aligned} 2(n+1)H(n)-4n &\leq T(n) \leq 2(n+1)H(n)-4n+2n \\ 2(n+1)H(n)-4n &\leq T(n) \leq \quad 2(n+1)H(n)-2n \end{aligned}$$

Es gilt jedoch noch $R(n) = \frac{T(n)}{n}$:

$$2\frac{n+1}{n}H(n)-4 \leq R(n) \leq 2\frac{n+1}{n}H(n)-2$$

Uns interessiert hier auch nur $R(n) \leq 2\frac{n+1}{n}H(n)-2$. Für große n gilt:

$$R(n) \leq 2\cdot H(n)-2$$

Dies gilt wegen:

$$\lim_{n\to\infty}\frac{n+1}{n} = \lim_{n\to\infty}\frac{n}{n}+\lim_{n\to\infty}\frac{1}{n} = 1+0 = 1$$

Wie schon bei Quicksort gesehen gilt:

$$H(n) = \ln n+\gamma$$

Dabei ist γ die Eulersche Konstante mit $\gamma \approx 0.57721....$ Also gilt:

$$\begin{aligned} R(n) &\leq 2 \cdot (\ln n + \gamma) - 2 \\ &\leq 2 \cdot \ln n + 2\gamma - 2 \\ &\leq 2 \cdot \ln 2 \cdot \log_2(n) - O(1) \\ &\leq 1.386 \cdot \log_2(n) - O(1) \end{aligned}$$

Also beträgt die durchschnittliche erwartete Suchzeit ca. $1.386 \cdot \log_2(n) - O(1)$, bzw. $O(\log_2(n))$.

3.4 2-3-Bäume

Die Klasse der 2-3-Bäume wurde 1970 von Hopcroft vorgestellt. 2-3-Bäume sollen dafür sorgen, daß der worst-case bei binären Bäumen nicht mehr auftritt. Sie sind balancierte Suchbäume.

- **Wann heißt ein Baum 2-3-Baum?**

Ein Baum heißt 2-3-Baum, wenn folgende Eigenschaften erfüllt sind:

1. Ein Knoten enhält entweder ein oder zwei Daten.
2. Knoten mit einem Datum x haben zwei Zeiger, wobei im linken Teilbaum nur die Daten sind, welche kleiner als x sind $(x' < x)$ und im rechten die Daten, die größer als x sind $(x < x'')$
3. Knoten mit zwei Daten x,y mit $x < y$ haben drei Zeiger, wobei links nur die Daten stehen, die kleiner als x sind $(x' < x)$, in der Mitte nur die Daten welche größer als x und kleiner als y sind $(x < x'' < y)$ und rechts nur Daten, die größer als y sind $(y < x''')$.
4. Die Zeiger die einen Knoten verlassen sind alle *NIL*-Zeiger oder alle keine *NIL*-Zeiger.
5. Alle Blätter haben gleiche Tiefe.

- **Was ist ein Nachteil bei 2-3-Bäumen?**

Ein Nachteil ist, daß Speicherplatz unter Umständen verschwendet wird, da jeder Knoten Platz für zwei Daten und drei Zeiger haben muß, die jedoch nicht immer genutzt werden. Außerdem müssen in einem Knoten unter Umständen zwei Vergleiche gemacht werden.

- **Welche Mindesttiefe bzw. Maximaltiefe haben 2-3-Bäume mit n Daten?**

Die Mindesttiefe beträgt $\lceil \log_3(n+1) \rceil - 1$ und die Maximaltiefe $\lceil \log_2(n+1) \rceil - 1$. Die größte Datenzahl bei einer Tiefe von d erreichen vollständige tenäre Bäume. Dann gibt es $2(1+3^1+3^2+...+3^d) = 3^{d+1}-1$ Daten. Also gilt:

$$\begin{aligned} 3^{d+1}-1 &\geq n \\ 3^{d+1} &\geq n+1 \\ d+1 &\geq \lceil \log_3(n+1) \rceil \\ d &\geq \lceil \log_3(n+1) \rceil - 1 \end{aligned}$$

Die kleinste Datenzahl wird von vollständigen binären Bäumen erreicht. Dann gibt es $1+2^1+2^2+...+2^d = 2^{d+1}-1$ Daten. Also gilt:

$$\begin{aligned} 2^{d+1}-1 &\leq n \\ 2^{d+1} &\leq n+1 \\ d+1 &\leq \lceil \log_2(n+1) \rceil \\ d &\leq \lceil \log_2(n+1) \rceil - 1 \end{aligned}$$

- **Wie funktioniert SEARCH(x) in einem 2-3-Baum?**

SEARCH(x):

Zuerst wird die Wurzel betrachtet. Hat ein Knoten ein Datum, so wird dieses mit x verglichen. Stimmen die Daten überein, können wir die Suche erfolgreich beenden. Ist x kleiner als das Datum im aktuellen Knoten, so suchen wir im linken Teilbaum weiter, sonst im rechten.

Wenn ein Knoten zwei Daten hat, verläuft die Suche ähnlich. Ist eines der beiden Daten gleich x, können wir die Suche wieder erfolgreich beenden. Seien die Daten im aktuellen Knoten y,z mit $x < z$. Ist $x < y$ so suchen wir im linken Teilbaum weiter, gilt $y < x < z$, so suchen wir im mittleren Teilbaum weiter und sonst im rechten Teilbaum $(z < x)$.

- **Wie funktioniert INSERT(x)?**

INSERT(x):

Zuerst schauen wir, ob x noch nicht im Baum vorhanden ist. Dabei wird der Suchpfad auf einem Stack gespeichert. Ist x im Baum vorhanden, so werden einfach die Daten geändert, sonst wird ein neuer Knoten eingefügt. Wir sind an einem Blatt angekommen und haben einen *NIL*-Zeiger, an den das einzufügende Datum eigentlich hin muß.

Ist das Blatt, an dem dieser *NIL*-Zeiger hängt, ein Knoten mit nur einem Datum y, so machen wir daraus einen Knoten mit zwei Daten x,y (wenn $x < y$) bzw. y,x (wenn $y < x$).

Hat das Blatt zwei Daten y,z (o.B.d.A. $y < z$), so müssen wir anders vorgehen. Gilt $x < y$, so stellen wir lokal die 2-3-Eigenschaft her, indem wir y in einen Zwei-Zeiger-Knoten packen, in dessen linken Teilbaum ein Zwei-Zeiger-Knoten mit dem Datum x ist und in dessen rechten Teilbaum ein Zwei-Zeiger-Knoten mit dem Datum z. Gilt $y < x < z$, so gehen wir ähnlich vor, nur daß x Wurzel wird mit y links und z rechts. Gilt $y < z < x$ wird z Wurzel, y steht links und x rechts.

Bei dieser Vorgehensweise gibt es nur ein Problem: Die Blätter sind jetzt nicht mehr gleich tief. Also schauen wir uns den Vorgänger der Wurzel an. Ist dies ein Knoten mit nur einem Datum, so packen wir die Wurzel unseres Problembaumes dort mit rein. Wie die Zeiger dabei zu setzen sind, dürfte klar sein. Ist es jedoch ein Knoten mit zwei Daten, so bauen wir diesen auf die gleiche Art und Weise um, wie wir es mit unserem Problembaum gemacht haben und setzen die Lösung mit seiner Wurzel fort. Dies machen wir so lange, bis wir an einen Knoten kommen, der nur ein Datum enthält oder wir an der Wurzel sind und dort das Problem auflösen können. Wenn wir das Problem an der Wurzel absorbieren, wächst die Tiefe des Baumes um 1. 2-3-Bäume wachsen also über die Wurzel.

- **Wie funktioniert DELETE(x)?**

DELETE(x):

Zuerst suchen wir das Datum x, welches wir löschen möchten. Sind wir dabei erfolgreich, können wir zwei Fälle unterscheiden: x steht in einem Blatt oder nicht. Wenn x nicht in einem Blatt steht, müssen wir dafür sorgen. Dazu suchen wir das Datum, welches in der Inorder-Reihenfolge vor x kommt. Also nehmen wir den ersten Zeiger links von x und gehen dann immer am Zeiger entlang der am weitesten rechts ist. Dies machen wir solange, bis wir auf einen *NIL*-Zeiger treffen. Im letzten Knoten vor dem *NIL*-Zeiger wählen wir das größere Datum y.

Wir haben jetzt dafür gesorgt, daß das zu löschende Datum in einem Blatt steht. Wieder können wir zwei Fälle unterscheiden:

1. Das Blatt, in dem x steht, ist ein Knoten mit zwei Daten. Wir löschen also x und einen Zeiger und alles ist in Ordnung.

2. Das Blatt, in dem x steht, ist ein Knoten mit einem Datum. Wir löschen das ganze Blatt, nur ist die 2-3-Eigenschaft im Vorgänger-Knoten (falls existent, sonst ist der ganze Baum gelöscht) verletzt.

Die Situation (2) formulieren wir ganz allgemein wie folgt: Wir haben einen Knoten v mit einem Datum x. Dieser hat mindestens zwei Zeiger, wobei ein Zeiger p auf einen Teilbaum zeigt, dessen Blätter zwar alle auf einer Höhe sind, der jedoch bezüglich des Restbaumes eine Ebene zu hoch hängt. Ein weiterer Zeiger q verläßt den Knoten v ebenfalls. Bei ihm ist alles in Ordnung. Jetzt unterscheidet man drei Fälle:

1. v ist ein Knoten mit einem oder zwei Daten und q zeigt auf einen Knoten der zwei Daten y,z mit $y < z$ enthält. Wir lassen jetzt y auf die Position von x wandern und die beiden Zeiger, die x umschlossen haben, zeigen jetzt auf x (der linke Zeiger) und auf z (der rechte Zeiger). x ist ein Knoten mit zwei Zeigern auf p und q_1 und z ist ebenfalls ein Knoten mit einem Datum und zwei Zeigern auf q_2 und q_3. Nach diesem Fall kann gestoppt werden, da das Problem komplett gelöst ist.

2. v enthält zwei Daten x' und x mit $x' < x$ und q zeigt auf den Knoten w mit nur einem Datum y und zwei Zeigern q_1 und q_2. Wir machen aus v einen Knoten mit einem Datum x' und zwei Zeigern, wobei der rechte auf einem Knoten mit zwei Daten x,y mit $x < y$ zeigt. Dieser Knoten hat drei Zeiger p, q_1, q_2.

3. v enthält nur x, und w nur y mit Zeigern q_1 und q_2. Weiter zeige der Zeiger r auf v. Wir machen aus v einen Knoten mit zwei Daten x,y und drei Zeigern p, q_1, q_2. Jetzt ist r ein Problemzeiger geworden und wir fahren analog fort.

Der günstigste Fall ist Fall 1, da dort das Problem nach einem Schritt gelöst wird. Wenn man also die Wahl hat, welchen Zeiger man als q-Zeiger betrachtet, sollte man, sofern möglich, einen Knoten mit zwei Daten wählen.

Die Rechenzeit ist proportional zur Tiefe, da man sich eventuell von einem Blatt bis zur Wurzel „hochkorrigiert".

- **Welche Laufzeit haben SEARCH, INSERT und DELETE in 2-3-Bäumen?**

$O(\log_2(n))$, da man sich höchstens den ganzen Baum herab- (SEARCH) bzw. heraufhangelt (INSERT, DELETE).

Im folgenden wird davon ausgegangen, daß die eigentlichen Informationen in den Blättern des 2-3-Baumes gespeichert sind. In den inneren Knoten wird das größte Datum des linken Teilbaumes gespeichert und falls existent das größte Datum des mittleren Teilbaumes. Weiter speichere jeder Baum seine Tiefe und sein größtes Datum.

- **Wie funktioniert CONCATENATE?**

CONCATENATE(T_1, T_2, T): Wir gehen davon aus, daß alle Daten in T_1 kleiner als die Daten in T_2 sind. Der Ergebnisbaum sei T. Wir unterscheiden zwei Fälle:

1. $d(T_1) = d(T_2)$. T ist eine Wurzel, die als Datum das größte Datum aus T_1 enthält und zwei Zeiger auf T_1 und T_2 hat. Das größte Datum in T, welches ja extra gespeichert wird, ist das größte Datum aus T_2.

2. $d(T_1) > d(T_2)$. Wir bilden einen Knoten T_2', der kein Datum enthält und einen Zeiger auf T_2. Jetzt gehen wir in T_1 $d(T_1) - d(T_2) - 1$ Schritte, wobei wir immer den rechtesten Zeiger benutzen. Dabei kommen wir an einen Knoten v. Es gibt wieder zwei Fälle:

   a) v enthält nur ein Datum x und zwei Zeiger. Wir verschmelzen v mit der Wurzel von T_2' zu einem Knoten mit zwei Daten x, y, wobei y das größte Datum aus T_1 ist. Das größte Datum aus T ist gleich dem größten Datum aus T_2.

   b) v enthält zwei Daten x, y und drei Zeiger auf T', T'' und T'''. Wir bilden einen Knoten mit einem Datum y und zwei Zeigern auf die Knoten mit den Daten x und z. Dabei ist z das größte Datum aus T_1. x hat zwei Zeiger auf T' und T'', z zwei Zeiger auf T''' und T_2. Jetzt hängt der Teilbaum y zu tief, diese Situation kennen wir jedoch von INSERT und lösen es auch genauso.

   Der Fall $d(T_2) > d(T_1)$ verläuft analog, nur daß man $d(T_2) - d(T_1) - 1$ Schritte in T_2 immer den linkesten Zeiger langläuft und T_1 links anfügt.

3.5 Bayer-Bäume

- **Wofür sind Bayer-Bäume (B-Bäume) gut?**

B-Bäume sind die Verallgemeinerung von 2-3-Bäumen. Bei 2-3-Bäumen können alle Daten im Kernspeicher gehalten werden, dies ist jedoch nicht immer der Fall. Für den Zugriff auf Daten, die nicht alle im Kernspeicher eines Rechners gehalten werden können, hat sich das Paging-Konzept durchgesetzt.

Die einzelnen Seiten werden dabei über Schlüsselwörter angesprochen, die wiederum selbst zu Seiten zusammenfasst sind. Ein Knoten enthält dabei alle Schlüsselwörter, die auf einer Seite zusammengefaßt worden sind. Dazu reichen 2 Daten pro Knoten nicht aus.

- **Wann heißt ein Baum B-Baum?**

Ein Baum heißt B-Baum der Ordnung m, wenn folgende Eigeschaften erfüllt sind:

1. Jeder Knoten enthält mindestens $\lceil \frac{m}{2} \rceil - 1$ und höchstens $m-1$ Daten. Die Wurzel bildet dabei eine Ausnahme, in ihr sind auch weniger als $\lceil \frac{m}{2} \rceil - 1$ Daten erlaubt. Die Daten in den Knoten sind dabei sortiert.

2. Knoten mit k Daten haben $k+1$ Zeiger, wobei alle Daten im ersten Zeiger links von einem Datum kleiner sind und im ersten Zeiger rechts größer.

3. Die Zeiger, die einen Knoten verlassen sind entweder alles *NIL*-Zeiger oder alle keine *NIL*-Zeiger.

4. Alle Blätter haben gleiche Tiefe.

2-3-Bäume sind also B-Bäume der Ordnung 3.

- **Wie funktioniert SEARCH(x) in B-Bäumen?**

Wir starten an der Wurzel und suchen das Datum mit binärer Suche, was möglich ist, da die Daten in den Knoten sortiert sind. Entweder wir finden das Datum x und können die Suche erfolgreich beenden oder wir finden eine Position, an der x eigentlich stehen müßte. An dieser Position ist auch ein Zeiger, der uns angibt, in welchem Knoten wir weitersuchen müssen.

- **Welche Laufzeit hat SEARCH(x) bei B-Bäumen?**

Die Laufzeit für ein SEARCH beträgt in etwa $\log_2 n$, wenn wir n Daten haben. In jedem Knoten brauchen wir im worst-case $\log_2((m-1)+1) = \log_2 m$ Vergleiche und die Tiefe des Baumes beträgt in etwa $\log_m n$. Also suchen wir $\log_m n \cdot \log_2 m = \log_2 m \cdot \log_m n = \log_2 n$.

- **Wie funktioniert INSERT(x) in einem B-Baum?**

Zuerst wird im B-Baum nach x gesucht. Finden wir es, so wird einfach nur das entsprechende Datum geändert. Sonst finden wir einen *NIL*-Zeiger, wo x eigentlich hin muß. Enthält der Knoten, zu dem der *NIL*-Zeiger gehört weniger als $m-1$ Daten, so wird das Datum einfach in den Knoten eingefügt. Enthält der Knoten genau $m-1$ Daten, so wird x einfach in den Knoten eingefügt. Jetzt haben wir einen Knoten mit m Daten und die B-Baum-Eigenschaft ist verletzt. Um sie wieder herzustellen bilden wir einen Teilbaum mit 3 Knoten:

Die Wurzel enthält das Datum $z_{\lceil \frac{m}{2} \rceil}$, wenn $z_1, ..., z_m$ die Daten im Knoten sind. Der linke Unterbaum enthält die Daten $z_1, ..., z_{\lceil \frac{m}{2} \rceil - 1}$ und der rechte die Daten $z_{\lceil \frac{m}{2} \rceil + 1}, ..., z_m$.

Der linke Knoten enthält genau richtig viele Daten. Auch im rechten Knoten ist die B-Baum-Bedingung erfüllt, denn er enthält $m - \lceil \frac{m}{2} \rceil = \lfloor \frac{m}{2} \rfloor \geq \lceil \frac{m}{2} \rceil - 1$ Daten.

Die Wurzel ist jetzt das neue Problem. Das Problem wird aufgelöst, wenn wir einen Knoten mit weniger als $m-1$ Daten finden oder es an der Wurzel auflösen, da dort weniger als $\lceil \frac{m}{2} \rceil - 1$ Daten erlaubt sind.

- **Wie funktioniert DELETE(x) bei B-Bäumen?**

Zuerst suchen wir das Datum x im Baum. Steht es in einem Blatt ist erstmal alles in Ordnung, sonst müssen wir dafür sorgen, daß es in einem Blatt steht und es dann löschen. Dafür tauschen wir es mit seinem Inorder-Vorgänger, was in B-Bäumen ähnlich verläuft wie in binären Suchbäumen. Dazu nehmen wir den Zeiger der links vom zu löschenden Datum steht und folgen dann immer den rechtesten Zeigern. Steht das Datum dann in einem Blatt gibt es mehrere Möglichkeiten:

1. Das Blatt enthält mehr als $\lceil \frac{m}{2} \rceil - 1$ Daten. Wir löschen einfach x und einen Zeiger und alles ist in Ordnung.

2. Das Blatt enthält genau $\lceil \frac{m}{2} \rceil - 1$ Daten und nach dem Löschen von x ein Datum zu wenig. Jetzt kann man wieder zwischen zwei Fällen unterscheiden:

   a) Das linke oder rechte Nachbarblatt hat mehr als $\lceil \frac{m}{2} \rceil - 1$ Daten. Es reicht eine einfache Datenrotation aus. Dabei wandert das Datum, welches die beiden Blätter trennt in das Blatt mit den $\lceil \frac{m}{2} \rceil - 2$ Daten (in dem gelöscht wurde) und das kleinste Datum im anderen Knoten wandert an die Position dieses Datums.

   b) Die beiden Nachbarblätter (eventuell gibt es auch nur eines) haben auch nur $\lceil \frac{m}{2} \rceil - 1$ Daten. Wir suchen uns eines der Blätter aus (wenn man überhaupt eine Wahl hat) und verschmelzen das Blatt, in dem gelöscht wurde, das Nachbarblatt sowie dem Datum, welches die Blätter trennt zu einem Knoten zusammen. Dieser Knoten enthält dann $\left(\lceil \frac{m}{2} \rceil - 2\right) + \left(\lceil \frac{m}{2} \rceil - 1\right) + 1 = 2\lceil \frac{m}{2} \rceil - 2 \leq m - 1$ Daten. Jedoch kann es jetzt sein, daß der Knoten, an dem dieser neue Knoten hängt jetzt ein Datum zu wenig hat, denn wir haben ja ein Datum (das Trenndatum) noch mit in den neuen Knoten gezogen. Hier lösen wir das Problem ebenso und spätestens an der Wurzel ist es gelöst.

3.6 AVL-Bäume

- **Was ist ein AVL-Baum?**

Ein AVL-Baum ist ein balancierter binärer Suchbaum. AVL-Bäume wurde von Adelson-Velskii und Landis entwickelt, nach denen sie auch benannt sind.

- **Wann ist ein binärer Suchbaum ein AVL-Baum?**

Ein binärer Suchbaum heißt AVL-Baum, wenn sich die Tiefen seines linken und rechten Teilbaumes um nicht mehr als 1 unterscheiden. Jeder Knoten v führt seinen Balancegrad $b(v)$ mit. Dabei gilt:

$$b(v) = d(T_l) - d(T_r)$$

In AVL-Bäumen gilt also für jeden Knoten $v : b(v) \in \{-1, 0, 1\}$.

- **Welche Mindesttiefe hat ein AVL-Baum mit n Daten?**

Die größte Datenzahl für eine Tiefe d wird von vollständigen binären Bäumen erreicht, also gilt:

$$\begin{aligned} 2^{d+1} - 1 &\geq n \\ 2^{d+1} &\geq n+1 \\ d+1 &\geq \lceil \log_2(n+1) \rceil \\ d &\geq \lceil \log_2(n+1) \rceil - 1 \end{aligned}$$

Also hat ein AVL-Baum mit n Daten die Mindesttiefe $\lceil \log_2(n+1) \rceil - 1$.

- **Welche Maximaltiefe hat ein AVL-Baum mit n Daten?**

Um diese Frage zu beantworten, müssen wir uns überlegen, wieviele Knoten man mindestens braucht, um eine bestimmte Tiefe zu erreichen. Denn diese Tiefe kann dann höchstens mit dieser Anzahl an Daten erreicht werden.

Ein Baum der Tiefe 0 hat mindestens ein Datum, ein Baum der Tiefe 1 mindestens 2. Sei $A(d)$ die Anzahl an Daten, die ein Baum die Tiefe d mindestens hat. Dann gilt:

$$A(d) = 1 + A(d-1) + A(d-2)$$

Ein Baum der Tiefe d hat mindestens eine Wurzel und einen Teilbaum der Tiefe $d-1$, also hat dieser Teilbaum $A(d-1)$ Knoten. Der andere Teilbaum kann unter Ausnutzung des Balancekriteriums die Tiefe $d-2$ haben, also $A(d-2)$ Knoten. Es ergibt sich also folgende Tabelle:

d	0	1	2	3	4	5	6	7	8
$A(d)$	1	2	4	7	12	20	33	54	88
$F(d)$	0	1	1	2	3	5	8	13	21

Es gilt anscheinend:

$$A(d) = F(d+3) - 1$$

Dies zeigen wir per Induktion über d:

Induktionsanfang:

$$\begin{aligned} A(0) &= 1 = 2 - 1 = F(3) - 1 \\ A(1) &= 2 = 3 - 1 = F(4) - 1 \end{aligned}$$

Induktionsvorraussetzung:

$$\begin{aligned} A(d) &= 1 + A(d-1) + A(d-2) \\ &= 1 + F(d+2) - 1 + F(d+1) - 1 \\ &= F(d+3) - 1 \end{aligned}$$

Wenn ein AVL-Baum also n Daten und Tiefe d hat, so gilt:

$$A(d) \leq n \Leftrightarrow F(d+3) - 1 \leq n \Leftrightarrow \boxed{F(d+3) \leq n+1}$$

Wie groß ist d denn nun? Wir ersetzen die rekursive Form der Fibonacci-Zahlen durch die geschlossene Form:

$$\begin{aligned} F(d+3) &\leq n+1 \\ \frac{1}{\sqrt{5}}\left(\Phi^{d+3} - \hat{\Phi}^{d+3}\right) &\leq n+1 \end{aligned}$$

Hierbei gilt $\Phi = \left(\frac{1+\sqrt{5}}{2}\right) \approx 1.618$ und $\hat{\Phi} = \left(\frac{1-\sqrt{5}}{2}\right) \approx -0.618$. Wie man sieht ist $\hat{\Phi} \leq 1$ und es gilt $\lim_{d\to\infty} \hat{\Phi}^{d+3} = 0$. Daher kann $\frac{1}{\sqrt{5}}\hat{\Phi}^{d+3}$ mit $\frac{1}{2}$ abgeschätzt werden, da es diesen Wert nie überschreitet. Also:

$$\begin{aligned} \frac{1}{\sqrt{5}}\Phi^{d+3} - \frac{1}{2} &\leq n+1 \\ \frac{1}{\sqrt{5}}\Phi^{d+3} &\leq n + \frac{3}{2} \\ \log_\Phi \frac{1}{\sqrt{5}} + d + 3 &\leq \log_\Phi \left(n + \frac{3}{2}\right) \\ d &\leq \log_\Phi n + O(1) \\ d &\leq \log_\Phi 2 \cdot \log_2 n + O(1) \\ d &\leq \frac{\ln 2}{\ln \Phi} \cdot \log_2 n + O(1) \\ d &\leq 1.4404 \cdot \log_2 n + O(1) \end{aligned}$$

Das heißt, daß ein AVL-Baum mit n Daten höchstens 44% tiefer ist als ein vollständiger binärer Baum mit n Daten. Dies ist auch der Grund, warum die Operationen SEARCH, INSERT und DELETE in $O(\log_2 n)$ durchführbar sind, wie wir später sehen werden.

- **Wie funktioniert SEARCH in einem AVL-Baum?**

SEARCH in einem AVL-Baum funktioniert genau wie SEARCH in normalen binären Bäumen.

- **Wie funktioniert INSERT in einem AVL-Baum?**

INSERT in einem AVL-Baum funktioniert genau wie INSERT bei normalen binären Bäumen. Nur kann es hier vorkommen, daß der Baum, bzw. ein Unterbaum außer Balance gerät, d.h. für einen Knoten v gilt $v \notin \{-1, 0, 1\}$. Diesen Knoten müssen wir dann wieder in Balance bringen. Wie wir gleich sehen werden, reicht auch eine Rebalancierungsaktion aus, um den kompletten Baum wieder in einen balancierten Zustand zu bringen.

Sei v der Knoten, an dem der Balancegrad nicht mehr stimmt. Da $b(v) \notin \{-1,0,1\}$ gilt, muß also einer der beiden Teilbäume von v jetzt mindestens um 2 Ebenen tiefer liegen als der andere. Jetzt kann man zwei Fälle unterscheiden, wobei wir im folgenden davon ausgehen, daß der rechte Teilbaum um eine Ebene gewachsen ist. Der Fall, daß der linke Teilbaum gewachsen ist, ist symmetrisch.

1. Sei der Knoten x die Wurzel des rechten Teilbaumes. Wenn das neue Datum im rechten Teilbaum von x eingefügt wurde, dann reicht eine einfache Linksrotation aus, um den Baum zu rebalancieren.

2. Wenn das Datum im linken Teilbaum von x eingefügt wurde, dann reicht eine einfache Rotation nicht mehr aus. Jedoch hilft eine Doppelrotation, in diesem Falle eine Rechts-Links-Rotation. Dabei wird am Knoten x eine Rechtsrotation durchgeführt und danach am Knoten v eine Linksrotation.

Also hilft beim Einfügen folgendes Schema:

Wo eingefügt?	Rotation
RR	L
RL	RL

Wo eingefügt?	Rotation
LL	R
LR	LR

- **Warum reicht eine Rotation oder Doppelrotation aus, um den Baum zu rebalancieren?**

Weil der Baum nach dem Rebalancieren wieder dieselbe Tiefe hat, die er vor der Einfügung hatte.

- **Wie funktioniert DELETE in einem AVL-Baum?**

Auch DELETE funktioniert prinzipiell wie DELETE in einem normalen binären Suchbaum, jedoch muß man auch hier unter Umständen rebalancieren. Hier reicht jedoch nicht immer nur eine Rotation oder Doppelrotation aus.

Sei wieder v der Knoten an dem der Balancegrad nicht mehr stimmt. Ein Teilbaum ist also wieder mindestens um zwei Ebenen tiefer. Der tiefere Baum heiße wieder x und hat zwei Teilbäume. Im folgenden wird davon ausgegangen, daß x der rechte Teilbaum von v ist. Wieder kann man ein paar Fälle unterscheiden:

1. Die Teilbäume von x haben gleiche Tiefe ($b(x) = 0$). Es reicht eine Linksrotation aus.

2. Der rechte Teilbaum von x ist eine Ebene tiefer als der linke. Diese Situation ist ähnlich wie beim Einfügen. Wir rebalancieren daher mit einer Linksrotation.

3. Der linke Teilbaum von x ist eine Ebene tiefer als der rechte. Wir rebalancieren auch hier mit einer Rechts-Links-Rotation.

Folgendes Schema:

Wo zu tief?	Rotation
RB	L
RR	L
RL	RL

Wo zu tief?	Rotation
LB	R
LL	R
LR	LR

Es ist anzumerken, daß nur in Fall (1) die Rebalancierung des ganzen Baumes abgeschlossen ist, da nur hier die alte Tiefe erhalten bleibt. In den anderen Fällen sinkt die Tiefe um eines, der Teilbaum verliert also an Tiefe. Daher kann sich eine Rebalancierungsaktion bis an die Wurzel fortsetzen.

3.7 Skiplisten

Bis jetzt erfüllen nur 2-3-Bäume alle Forderungen, die an dynamische Dateien in Kapitel 3.1 gestellt wurden. Zwar werden die meisten Operationen in $O(\log_2 n)$ ausgeführt, jedoch verstecken sich in dieser Notation relativ große konstante Faktoren, die durch die ständige Rebalancierung entstehen. Skiplisten stellen ebenfalls alle Operationen zur Verfügung, sind jedoch etwas effizienter.

Hat die Sache einen Haken? Ja, aber nur einen kleinen oder vielleicht sollte man besser sagen: einen unwahrscheinlichen. Skiplisten sind randomisierte Datenstrukturen, d.h. ihr Verhalten kann sehr schlecht sein, dieser Fall ist jedoch auch sehr unwahrscheinlich.

- **Was sind Skiplisten?**

Wenn man in einer normalen, geordneten linearen Liste L ein Objekt sucht, so dauert dies $O(LENGTH(L))$, da das gesuchte Datum im Durchschnitt in der Mitte steht. In einem Array kann man dank binärer Suche ein Objekt in $O(\log_2 n)$ finden.

Skiplisten ahmen die binäre Suche in einem Array nach, indem sie Hilfszeiger benutzen. Jedes Objekt hat eine bestimmte Höhe, die angibt, wieviele Zeiger auf das Objekt zeigen. Desweiteren gibt es noch einen Listenanfang, der soviele Zeiger hat, wie die Höhe ist und ein Listenende.

Jedes Objekt inklusive Listenanfang hat in Höhe h einen Zeiger, der auf das nächste Objekt zeigt, dessen Höhe mindestens h ist. Ist ein solches Objekt nicht mehr vorhanden, so zeigt der Zeiger auf das Listenende.

- **Wie funktioniert SEARCH(x) in Skiplisten?**

SEARCH(x):

Wir beginnen am Listenanfang in Höhe h, wenn h die höchste Höhe ist, die ein Objekt hat. Wir betrachten das Objekt, auf das der Zeiger in Höhe h zeigt. Allgemein kann man drei Fälle unterscheiden:

1. Der Zeiger zeigt auf das gesuchte Objekt. Suche erfolgreich beenden.
2. Der Zeiger zeigt auf ein zu großes Datum oder auf das Listenende. In Höhe $h-1$ vom letzten Objekt aus weitersuchen. Ist man schon ganz unten wird die Suche erfolglos abgebrochen.
3. Der Zeiger zeigt auf ein zu kleines Objekt. Suche vom Objekt aus in Höhe h das nächste Objekt, welches Höhe h hat.

- **Wie funktioniert INSERT(x)?**

INSERT(x):

Zuerst suchen wir x. Ist x schon vorhanden, werden die Daten aktualisiert. Sonst muß ein neues Objekt für x eingefügt werden.

Wir würfeln zuerst eine Höhe für das Objekt aus. Dies machen wir nach dem Zufallsprinzip. Und zwar werfen wir eine Münze so oft, bis sie das erste Mal auf Kopf fällt. Fällt die Münze nach h Würfen zum ersten Mal auf Kopf, so bekommt das Objekt die Höhe h. Ist h größer als die bisherige höchste Höhe, so werden entsprechend viele Listen am Listenanfang gebildet, die sofort auf das neue Objekt zeigen und von dort auf das Listenende. Weiter merken wir uns bei der Suche nach x alle Zeiger, die auf Objekte gezeigt haben, die größer als x sind. Diese Zeiger zeigen nun direkt auf x

und die Zeiger von x zeigen auf die Objekte, auf die die abgespeicherten Zeiger zuvor gezeigt haben.

- **Wie hoch ist ein Objekt durchschnittlich?**

Einen worst-case für die Höhe gibt es nicht, da ein Objekt beliebig hoch werden kann. Es kann jedoch der Erwartungswert angegeben werden, der besagt, wie hoch ein Objekt im Durchschnitt ist. Die Wahrscheinlichkeit, daß ein Objekt die Höhe h bekommt, beträgt $\left(\frac{1}{2}\right)^h$. Somit beträgt der Erwartungswert:

$$\sum_{h=1}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \sum_{h=1}^{\infty} h \cdot 2^{-h} = 2$$

Ein Objekt hat also durchschnittlich die Höhe 2.

- **Wie funktioniert DELETE(x) in einer Skipliste?**

DELETE(x):

Dazu suchen wir das Objekt, daß kleiner als x ist, jedoch das größte Objekt unter den Objekten, die kleiner als x sind. Bei dieser Suche finden wir h Zeiger, die direkt auf x zeigen, wenn x die Höhe h hat. Diese Zeiger müssen nun in den entsprechenden Höhen direkt auf die Nachfolger von x zeigen. Dann kann das Objekt x gelöscht werden. Dabei kann die Höhe der Skipliste (der Anfangsliste) sinken.

- **Wie funktionieren CONCATENATE und SPLIT bei Skiplisten?**

Im folgenden wird davon ausgegangen, daß bei CONCATENATE alle Objekte in der einen Skipliste kleiner sind als in der anderen und daß bei SPLIT das Splitobjekt auch tatsächlich existiert.

CONCATENATE:

Bei CONCATENATE suchen wir in der Skipliste mit den kleineren Elementen alle Zeiger auf das Listenende. Diese Zeiger zeigen nun auf das erste Objekt, sofern sie nicht höher liegen als das erste Objekt. In diesem Fall zeigen sie direkt auf das Listenende der neuen Liste.

SPLIT:

Hier suchen wir alle Zeiger, die auf Objekte zeigen, die größer sind als das Splitobjekt. Wenn unsere alte Liste die Höhe h hatte (Anfangsliste hat Höhe h), dann erschaffen wir ein neues Endobjekt und Anfangsobjekt jeweils mit Höhe h. Die Zeiger die wir gefunden haben setzen wir nun direkt auf das Endobjekt und vom neuen Anfangsobjekt aus starten die Zeiger auf die Elemente der zweiten Liste. Unter Umständen sinkt die Höhe in einer der beiden Listen.

- **Welche Höhe hat die Anfangsliste im worst-case?**

Auch hier gilt, daß es einen worst case nicht gibt, da die Anfangsliste beliebig hoch werden kann. Man kann jedoch einen Erwartungswert berechnen, der quasi aussagt, wie hoch die Liste im Durchschnitt wird.

Wie groß ist also die Wahrscheinlichkeit, daß eine Liste mit n Daten mindestens die Höhe h erreicht. Wir schätzen diese Wahrscheinlichkeit ab durch:

$$Prob(H(n) \geq h) = \min\left\{1, n \cdot \left(\frac{1}{2}\right)^{h-1}\right\}$$

Diese Formel bedeutet quasi, daß kleine Höhen auf jeden Fall vorkommen und nur große Höhen immer unwahrscheinlicher werden. Dies ist eine relativ grobe

Abschätzung nach oben, jedoch wollen wir ja auch eine obere Schranke nachweisen. Jetzt überlegen wir uns erstmal, für welche h als Wahrscheinlichkeit 1 gewählt wird. Es gilt doch dann:

$$\begin{aligned} n \cdot \left(\frac{1}{2}\right)^{h-1} &\geq 1 \\ n &\geq 2^{h-1} \\ \lceil \log_2 n \rceil &\geq h-1 \\ h-1 &\leq \lceil \log_2 n \rceil \\ h &\leq \lceil \log_2 n \rceil + 1 \\ h &\leq \lfloor \log_2 n \rfloor + 2 \end{aligned}$$

Also schätzen wir hier ab, daß bei n Daten die Höhen $\lfloor \log_2 n \rfloor + 2$ auf jeden Fall vorkommen. Natürlich ist dies nicht die ganze Wahrheit, jedoch reicht es für unsere Abschätzung. Ein Beispiel noch. Angenommen wir haben 1000 Daten. Welche Höhen nehmen wir jetzt als 100%-ig an? Die Höhen $\lfloor \log_2 1000 \rfloor + 2 = 9 + 2 = 11$. Wir gehen also davon aus, daß bei 1000 Daten die Höhen bis 11 auf jeden Fall vorhanden sind. Wie schon gesagt, muß dies nicht sein, es könnten zum Beispiel mit sehr, sehr kleiner Wahrscheinlichkeit nur Objekte der Höhe 1 entstehen.

Doch kommen wir nun zum Erwartungswert der Höhen. Der Erwartungswert ist folgender:

$$\begin{aligned} E(H(n)) &= \sum_{h=1}^{\infty} h \cdot Prob(H(n) = h) \\ &= Prob(H(n) = 1) + Prob(H(n) = 2) + Prob(H(n) = 3) + \ldots \\ &\quad + Prob(H(n) = 2) + Prob(H(n) = 3) + \ldots \\ &\quad + Prob(H(n) = 3) + \ldots \\ &= Prob(H(n) \geq 1) \\ &\quad + Prob(H(n) \geq 2) \\ &\quad + Prob(H(n) \geq 3) \\ &\quad + \ldots \end{aligned}$$

Wir sortieren die Summanden also um, damit wir eine Formel erhalten, die wir schon kennen. Es gilt also:

$$E(H(n)) = \sum_{h=1}^{\infty} Prob(H(n) \geq h)$$

Wie wir oben schon gesehen haben, gilt für $h \leq \lfloor \log_2 n \rfloor + 2$, daß $Prob(H(n) \geq h) = 1$ ist. Also können wir die ersten $\lfloor \log_2 n \rfloor + 2$ Summanden durch 1 abschätzen. Die

restlichen durch $\left(\frac{1}{2}\right)^{h-1}$. Daraus folgt:

$$\begin{aligned} E(H(n)) &= \sum_{h=1}^{\lfloor \log_2 n \rfloor + 2} 1 + \sum_{h=\lfloor \log_2 n \rfloor + 3}^{\infty} \left(\frac{1}{2}\right)^{h-1} \\ &\leq \lfloor \log_2 n \rfloor + 2 + \sum_{h=1}^{\infty} \left(\frac{1}{2}\right)^{h} \\ &= \lfloor \log_2 n \rfloor + 2 + 1 \\ &= \lfloor \log_2 n \rfloor + 3 \end{aligned}$$

- **Wie groß ist die erwartete Anzahl an Zeigern?**
  Jedes Objekt hat durchschnittlich die Höhe 2. Also brauchen wir für jedes Objekt schon mal durchschnittlich 2 Zeiger. Die erwartete Höhe der Anfangsliste ist, wie wir oben gesehen haben $\lfloor \log_2 n \rfloor + 3$. Also brauchen wir für die Anfangsliste nochmal so viele Zeiger. Daraus ergibt sich der Erwartungswert $E(Z(n))$:

$$E(Z(n)) \leq 2n + \lfloor \log_2 n \rfloor + 3$$

- **Welchen Platzbebarf kann man bei Skiplisten erwarten?**
  Der erwartete Platzbedarf bei Skiplisten ist $O(n)$, da gilt $2n + \lfloor \log_2 n \rfloor + 3 \in O(n)$.

4 Entwurfsmethoden für Algorithmen

4.1 Greedy Algorithmen

- **Was ist ein Greedy Algorithmus?**

Die Methode der Greedy Algorithmen ist auf Optimierungsprobleme anwendbar, bei denen die möglichen Lösungen aus mehreren Teillösungen bestehen. Das verfahren der Auswahl, welche Teillösung dabei genommen wird, ist simpel. Es wird immer gierig (greedy) die Teillösung gewählt, die die Gesamtlösung am meisten erhöht (bzw. minimiert, wie bei Kruskal). Dabei wird jedoch nicht planvoll an die Zukunft gedacht.

- **Welche Möglichkeiten der Güte der Gesamtlösung gibt es für Greedy Algorithmen?**

1. Wir erhalten stets eine optimale Lösung. (Kruskal)
2. Wir erhalten nicht immer eine optimale Lösung, jedoch weichen nicht optimale Lösungen nur wenig von der optimalen Lösung ab.
3. Wir können beliebig schlechte Lösungen erhalten. (Münzwechselproblem; Traveling Salesman Problem)

- **Worin besteht das Münzwechselproblem?**

Das Münzwechselproblem besteht darin, einen gegebenen Betrag N mit möglichst wenig Münzen zusammenzusetzen. Dabei gibt es k verschiedene Münzen zu $n_1,..,n_k$ Einheiten. Damit alle Beträge bezahlbar sind, muß $n_k = 1$ gelten und o.B.d.A. gelte $n_1 > ... > n_k$.

- **Gebe einen Algorithmus für das Münzwechselproblem an!**

1. $W = N$
2. for $i = 1,...,k$: Wähle $\left\lfloor \frac{W}{n_i} \right\rfloor$ Münzen mit n_i Einheiten und setze $W = W - n_i \cdot \left\lfloor \frac{W}{n_i} \right\rfloor$.

Dieser Algorithmus terminiert auf jeden Fall, da $n_k = 1$ ist. Die Laufzeit ist $O(k)$, hängt also nur von der Anzahl verschiedener Münzen ab.

- **Zeige, daß für diesen Algorithmus zum Münzwechselproblem beliebig schlechte Lösungen möglich sind.**

Wir wählen $k = 3$ mit $n_3 = 1, n_2 > 3$ beliebig und $n_1 = 2n_2 + 1$. Wir wollen eine Lösung für $N = 3n_2$ finden. Offensichtlich würden drei Münzen vom Wert n_2 reichen. Der Algorithmus wählt jedoch eine Münze vom Wert n_1, wobei $W = n_2 - 1$ übrigbleibt. Also können wir keine Münze vom Wert n_2 ausgeben und wählen daher $n_2 - 1$ Münzen vom Wert n_3. Über n_2 kann man diese Lösung beliebig schlecht machen, also gehört das Münzwechselproblem in die Kategorie (3) (s.o.).

4.2 Dynamische Programmierung

- **Wie ist die Dynamische Programmierung motiviert?**

Bei vielen Divide-And-Conquer-Algorithmen weiß man, wo man das größere Problem in die Teilprobleme teilen muß. Dies ist jedoch nicht immer der Fall und man muß unter Umständen alle Zerlegungen durchprobieren. Jedoch entstehen bei diesem Ansatz leicht exponentielle Kosten:

Wenn wir ein Problem $[1...n]$ haben, so gibt es $n-1$ mögliche Zerlegungen dieses Problems. Nämlich für $i = 1,...,n-1$ in die Probleme $[1...i]$ und $[i+1...n]$. Angenommen die Lösung eines Problems der Größe 1 kostet eine Einheit und die Aufteilung selbst ist umsonst. Dann gilt für die Rechenzeit:

$$R(1) = 1 \text{ und } R(n) = \sum_{i=1}^{n-1} (R(i) + R(n-i)) = 2\sum_{i=1}^{n-1} R(i)$$

Weiter gilt:

$$2\sum_{i=1}^{n-1} R(i) \geq 2R(n-1)$$

Wir betrachten jetzt also jeweils nur den letzten Summanden in der Summe. Es gilt also $R(n) \geq 2R(n-1)$ für $n \geq 2$ und damit $R(n) \geq 2^n$. Die Rechenzeit ist somit exponentiell.

Jedoch werden bei uns einige Probleme sehr oft gelöst. So entsteht das Problem $[1...i]$ auch, wenn man ein Problem $[1...j]$ bildet mit $j > i$ als Teilungspunkt und dieses weiter aufteilt. Es gibt jedoch nur $\binom{n}{2} + n$ verschiedene Probleme $[i...j]$ mit $1 \leq i \leq j \leq n$.

- **Zeige, daß es bei der dynamischen Programmierung nur $\binom{n}{2} + n$ verschiedene Probleme gibt!**

Wenn man in $[1...n]$ einen Punkt fix nimmt, gibt es bis n nur $(n-i)+1$ Probleme. Nimmt man z.B. 1 fix, so gibt es $(n-1)+1 = n$ Probleme, nämlich $[1,1],[1,2],...,[1,n]$. Läßt man das i von 1 bis n laufen erhält man also die Gesamtanzahl an verschiedenen Problemen:

$$\sum_{i=1}^{n}[(n-i)+1] = \sum_{i=1}^{n}(n-i) + \sum_{i=1}^{n} 1 = \sum_{k=0}^{n-1} k + n = \frac{(n-1)n}{2} + n = \binom{n}{2} + n$$

Es ist also intelligenter, die Teilprobleme $[i...j]$ nach wachsendem $(j-i)$ zu lösen und die Teilprobleme, wenn sie zur Lösung größerer Probleme gebraucht werden, per Table-Look-Up bereitzustellen.

Allgemein löst man also erst kleine Probleme und baut aus diesen eine große zusammen.

- **Was ist das All-Pairs-Shortest-Path-Problem?**

Beim All-Pairs-Shortest-Path-Problem geht es darum, zwischen allen Knoten in einem Graphen $G = (V,E)$ mit Kantenbewertung $c : E \to \mathbb{R}^+$ die jeweils kürzesten Wege zu finden. Im folgenden sei $V = \{1,...,n\}$.

Um von einem Knoten i zu einem Knoten j zu kommen sind dabei alle anderen Knoten als mögliche Zwischenpunkte erlaubt. Dies ist auch der Schlüssel zum Algorithmus. Die kleinsten Probleme sind die Kosten um direkt von i nach j zu gelangen.

$w_{i,j}^{(l)}$ seien allgemein die Kosten um von i nach j zu gelangen, wenn nur Knoten aus $\{1,...,l\}$ als Zwischenknoten erlaubt sind. Demnach ist $w_{i,j}^{(0)} = c(i,j)$. Wir speichern im folgenden diese Werte $w_{i,j}^{(l)}$ in einer quadrtischen Matrix $W^{(l)}$.

Der Algorithmus arbeitet wie folgt: Zu Beginn lassen wir keinen anderen Knoten als Zwischenknoten zu. Die Matrix $W^{(0)}$ ist also identisch mit der Adjazenzmatrix zu G. Daraus bauen wir die Matrix $W^{(1)}$ wie folgt:

$$w_{i,j}^{(l+1)} = \min\{w_{i,j}^{(l)}, w_{i,l+1}^{(l)} + w_{l+1,j}^{(l)}\}$$

Entweder ist der Weg über den Knoten $l+1$ also ein Umweg oder die Wege vom Knoten i zum Knoten $l+1$ und von $l+1$ zu j sind addiert billiger als der direkte Weg.

Beispiel:
Der Graph entspricht natürlich $W^{(0)}$ und es wird empfohlen ihn aufzumalen.

$$W^{(0)} = \begin{pmatrix} \infty & 1 & 10 & 5 & \infty \\ \infty & \infty & 1 & 3 & \infty \\ \infty & \infty & \infty & 1 & 7 \\ \infty & \infty & \infty & \infty & 3 \\ 100 & \infty & \infty & \infty & \infty \end{pmatrix}$$

$$W^{(1)} = \begin{pmatrix} \infty & 1 & 10 & 5 & \infty \\ \infty & \infty & 1 & 3 & \infty \\ \infty & \infty & \infty & 1 & 7 \\ \infty & \infty & \infty & \infty & 3 \\ 100 & \boxed{101} & \boxed{110} & \boxed{105} & \infty \end{pmatrix}$$

$$W^{(2)} = \begin{pmatrix} \infty & 1 & \boxed{2} & \boxed{3} & \infty \\ \infty & \infty & 1 & 3 & \infty \\ \infty & \infty & \infty & 1 & 7 \\ \infty & \infty & \infty & \infty & 3 \\ 100 & 101 & \boxed{102} & \boxed{104} & \infty \end{pmatrix}$$

$$W^{(3)} = \begin{pmatrix} \infty & 1 & 2 & 3 & \boxed{9} \\ \infty & \infty & 1 & 3 & \boxed{8} \\ \infty & \infty & \infty & 1 & 7 \\ \infty & \infty & \infty & \infty & 3 \\ 100 & 101 & 102 & \boxed{103} & \infty \end{pmatrix}$$

$$W^{(4)} = \begin{pmatrix} \infty & 1 & 2 & 3 & \boxed{6} \\ \infty & \infty & 1 & 3 & \boxed{6} \\ \infty & \infty & \infty & 1 & \boxed{4} \\ \infty & \infty & \infty & \infty & 3 \\ 100 & 101 & 102 & 103 & \infty \end{pmatrix}$$

$$W^{(5)} = \begin{pmatrix} \infty & 1 & 2 & 3 & 6 \\ \boxed{106} & \infty & 1 & 3 & 6 \\ \boxed{104} & \boxed{105} & \infty & 1 & 4 \\ \boxed{103} & \boxed{104} & \boxed{105} & \infty & 3 \\ 100 & 101 & 102 & 103 & \infty \end{pmatrix}$$

Hier sind die Werte die sich gegenüber der Vorgänger-Matrix geändert haben eingekästelt. Desweiteren gilt, daß in $w_{i,j}^{(l)}$ die l-te Zeile und Spalte genau wie bei der Vorgänger-Matrix bleibt, da immer $w_{x,x}^{(l)} = \infty$ gilt, es also keine Verbesserung geben kann.

Der Algorithmus hat eine Laufzeit von $O(n^3)$, da man für jedes $W^{(l)}$ quadratisch viele Additionen und Vergleiche macht. Die $W^{(l)}$-Matrizen werden genau n-mal gebildet. Also gilt: $2 \cdot n^2 \cdot n = 2n^3 = O(n^3)$.

4.3 Branch-And-Bound Methoden?

- **Wie funktionieren Branch-And-Bound Methoden?**

Branch-And-Bound Methoden bestehen im allgemeinen aus drei Modulen:

1. Upper Bound: Berechnung einer (möglichst guten) oberen Schranke.
2. Lower Bound: Berechnung einer (möglichst guten) unteren Schranke.
3. Branching: Zerlegung des Problems (der Lösung) in disjunkte Teilprobleme (Teillösungen), die vom gleichen Typ sind und „rekursiv" gelöst werden können.

- **Gebe ein Beispiel für den Branch-And-Bound Ansatz an!**

Das Knapsack-Problem (KP) läßt sich mit der Branch-And-Bound Methode lösen:

1. Upper Bound: Berechne eine Lösung mit dem Algorithmus, der die Effizienz der Objekte berechnet und ein Zerschneiden des letzten Objektes, welches nicht mehr ganz in den Rucksack paßt, zuläßt. Die Lösung ist eine sichere Obere Schranke.
2. Lower Bound: Berechne eine Lösung mit dem Algorithmus, der die Effizienz der Objekte berechnet und sie dann sukzessive in den Rucksack packt. Diesmal jedoch ohne die Objekte zu zerschneiden. Dieser Algorithmus liefert eine sichere untere Schranke.
3. Branching: Betrachte das Objekt, welches bei der Berechnung der Upper Bound zerteilt wird. Zerlege das Problem in zwei Probleme, bei denen das Objekt einmal auf jeden Fall in den Rucksack gepackt wird und einmal auf keinen Fall.

- **Welche Datenstruktur wird bei Branch-And-Bound Methoden benutzt?**

Als Datenstruktur wird ein sogenannter Branch-And-Bound Baum (BBB) benutzt. An den Knoten stehen die Teilprobleme. Die Blätter stellen eine disjunkte Zerlegung des Gesamtproblems dar. Allgemein arbeitet man wie folgt auf dem Baum:

1. Initialisiere den BBB mit einem Knoten, der das Gesamtproblem darstellt.
2. Berechne die obere Schranke U und untere Schranke L als Maximum aller oberen und unteren Schranken in den Blättern des BBB.
3. Ist $L = U$ so hat man die optimale Lösung gefunden.
4. Ist $L < U$, wird ein Blattproblem in seine Teilprobleme (bei KP sind es zwei) zerlegt und zu Schritt (2) gegangen.

Im Falle des KP haben wir für den dritten Fall bei der Berechnung der unteren Schranke die Lösung gefunden.

- **Woraus beziehen die Branch-And-Bound Methoden ihre Effizienz?**

Angenommen wir haben ein Maximierungsproblem wie KP. Wenn in einem Teilproblem die obere Schranke schlechter ist als die beste untere Schranke, so brauchen wir dieses Teilproblem garnicht mehr betrachten.

Desweiteren muß man den Algorithmus nicht immer bis zum Ende laufen lassen, wenn die Lösung nicht optimal sein muß, sondern ruhig um einen bestimmten Wert von der optimalen Lösung abweichen darf. Die optimale Lösung ist dann höchstens $U - L$ besser als die gefundene.

4.4 Eine allgemeine Analyse des Divide-And-Conquer Ansatz

Hier wird auf Fragen verzichtet, da eh fast nur gerechnet wird. Wir gehen im folgenden davon aus, daß wir ein Problem der Größe n in a Teilprobleme der Größe $\lceil \frac{n}{b} \rceil$ bzw. $\lceil \frac{n}{b} \rceil - 1$ teilen. Die Arbeit die vor jeder Teilung anfällt sei cn. Bei Quicksort ist diese Arbeit z.B. $n-1$ (für die Vergleiche). $R(n)$ sei die Rechenzeit dieses DAC-Ansatzes. Insbesondere ist $R(1) = c \cdot 1 = c$. Im folgenden sei $n = b^k$, damit wir uns nicht mit gerundeten Zahlen herumschlagen müssen. Also ist $k = \log_b n$. Gegeben ist also:

$$R(1) = c \text{ und } R(n) = aR\left(\frac{n}{b}\right) + cn \text{ für } n \geq 2$$

Also gilt für $n = b^k$ und $a, b, c > 0$:

$$\begin{aligned} R(b^k) &= aR(b^{k-1}) + cb^k \\ &= a(aR(b^{k-2}) + cb^{k-1}) + cb^k \\ &= a^2R(b^{k-2}) + acb^{k-1} + cb^k \\ &= a^3R(b^{k-3}) + a^2cb^{k-2} + acb^{k-1} + cb^k \\ &= a^kR(1) + a^{k-1}cb + a^{k-2}cb^2 + \ldots + a^2cb^{k-2} + acb^{k-1} + cb^k \\ &= a^kc + a^{k-1}cb + a^{k-2}cb^2 + \ldots + a^2cb^{k-2} + acb^{k-1} + cb^k \end{aligned}$$

Es gilt also:

$$R(b^k) = c\left(a^k + a^{k-1}b + a^{k-2}b^2 + \ldots + a^2b^{k-2} + ab^{k-1} + b^k\right)$$

Jetzt kann man drei Fälle unterscheiden:

1. $\underline{a < b}$. Wir klammern b^k aus:

$$\begin{aligned} R(b^k) &= cb^k\left(\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \frac{a^{k-2}}{b^{k-2}} + \ldots + \frac{a^2}{b^2} + \frac{a}{b} + 1\right) \\ &= cb^k\left(1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2 + \ldots + \left(\frac{a}{b}\right)^{k-1} + \left(\frac{a}{b}\right)^k\right) \end{aligned}$$

   In der Klammer dieser Formel steht jetzt eine geometrische Reihe. Jetzt wird auch klar, warum einen der Mathematik-Professor mit solchen Dingen gequält hat. Es gilt im allgemeinen:

$$\sum_{i=0}^{k} q^i = 1 + q + q^2 + \ldots + q^k = \frac{1 - q^{k+1}}{1 - q}$$

   Für $|q| < 1$ gilt:

$$\lim_{k \to \infty} \frac{1 - q^{k+1}}{1 - q} = \frac{1}{1 - q}$$

   Die geometrische Reihe konvergiert also für $-1 < q < 1$. Die Rolle des q spielt bei uns $\frac{a}{b}$ und dieser Bruch ist nach Vorraussetzung kleiner als 1. Bei uns steht also:

$$R(b^k) = cb^k \cdot \sum_{i=0}^{k} \left(\frac{a}{b}\right)^i < cb^k \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b}\right)^i = cb^k \cdot \frac{1}{1 - \frac{a}{b}}$$

c und $\frac{1}{1-\frac{a}{b}}$ sind jedoch Konstanten also gilt:

$$R(n) = cn \cdot \frac{1}{1-\frac{a}{b}} = \Theta(n)$$

2. $\underline{a = b}$. in diesem Falle ergibt die obige Formel folgendes:

$$R(b^k) = cb^k \left(1 + 1^1 + 1^2 + ... + 1^{k-1} + 1^k\right) = cb^k(k+1)$$

Wir ersetzen b^k und k:

$$R(n) = cn(\log_b n + 1) = \Theta(n \log n)$$

3. $\underline{a > b}$. Dieser Fall verläuft ähnlich wie Fall 1, nur das diesmal a^k ausgeklammert wird:

$$R(b^k) = ca^k \left(1 + \frac{b}{a} + \left(\frac{b}{a}\right)^2 + ... + \left(\frac{b}{a}\right)^{k-1} + \left(\frac{b}{a}\right)^k\right)$$

Also gilt hier logischerweise:

$$R(b^k) = ca^k \cdot \sum_{i=0}^{k} \left(\frac{b}{a}\right)^i < ca^k \cdot \sum_{i=0}^{\infty} \left(\frac{b}{a}\right)^i = ca^k \cdot \frac{1}{1-\frac{b}{a}}$$

Wieder sind c und $\frac{1}{1-\frac{b}{a}}$ Konstanten, so daß wir nur noch a^k anders ausdrücken müssen. Wir formen es wie folgt um:

$$a^k = a^{\log_b n} = \left(n^{\log_n a}\right)^{\log_b n} = n^{\log_n a \cdot \log_b n} = n^{\log_b n \cdot \log_n a} = n^{\log_b a}$$

Folglich:

$$R(n) = c \cdot \frac{1}{1-\frac{b}{a}} \cdot n^{\log_b a} = \Theta(n^{\log_b a})$$

5 Algorithmische Geometrie

5.1 Plane Sweep Algorithmen

• **Gebe einen Algorithmus an, der entscheidet, ob sich n horizonale und n vertikale Liniensegmente in einer Ebene schneiden!**
Wenn wir naiv an das Problem rangehen, vergleichen wir einfach alle Liniensegmente miteinander. Ein solcher Algorithmus hätte jedoch eine Laufzeit von $\Theta(n^2)$, da die Anzahl Vergleiche gleich $\binom{n}{2}$ wäre.

Wir benutzen daher eine **Plane Sweep Methode**, die auch einige der Datenstrukturen nutzt, die wir bis jetzt kennengelernt haben. Gegeben seien also n horizontale und n vertikale Liniensegmente. Was ist das besondere an den horizontalen Segmenten? Die horizontalen Segmente haben eine konstante y-Koordinate z_i und die x-Koordinate ist ein Intervall $[x_i, y_i]$. Bei den vertikalen Segmenten ist es anders herum. Was folgt nun daraus, wenn wir eine Sweep Line, also eine vertikal ausgerichtete Linie, über die Ebene wandern lassen?

1. Horizontale Segmente schneiden die Sweep Line während eines Zeitintervalls, da die Sweep Line sich zeitlich gesehen entlang der x-Achse bewegt, bleiben jedoch in der y-Koordinate dabei konstant.

2. Vertikale Segmente schneiden die Sweep Line in genau einem Zeitpunkt und die y-Koordinate ist ein Intervall.

Wir nutzen im folgenden Algorithmus aus, daß der Schnitt eines vertikalen Segmentes mit einem horizontalen Segmentes zu genau einem Zeitpunkt stattfindet. Für den Algorithmus brauchen wir einen **AVL-Baum**, dessen Objekte (insofern er welche enthält) durch eine **doppelt verkettete Liste** miteinander verknüpt sind.

Als Eingabe erhält der Algorithmus n horzontale Segmente $[x_i, y_i] \times z_i$ und n vertikale Segmente $a_i \times [b_i, c_i]$.

1. Wichtig sind die sogenannten **Ereignispunkte** x_i, y_i und a_i, bei denen entweder ein horizontales Liniensegment beginnt, endet oder ein Schnitt mit einem vertikalen Element überhaupt erst möglich wird. Daher **sortieren** wir diese x-Koordinaten erst einmal. Die Sweep Line betrachtet dann nur diese Ereignispunkte, da nur hier etwas Neues passiert. Die Laufzeit ist hier $O(n \log_2 n)$.

2. **Initialisiere** einen AVL-Baum, der alle y-Koordinaten von horizontalen Elementen enthält, die sich momentan mit der Sweep Line schneiden. Die Laufzeit ist $O(1)$.

3. **Durchlaufe** die sortierte Folge:

   a) Falls der Anfang eines horizontalen Segmentes (x_i) gefunden wird, füge die y-Koordinate dieses Segmentes in den AVL-Baum ein.

   b) Falls das Ende eines horizontalen Segmentes (y_i) gefunden wird, so lösche die y-Koordinate dieses Segmentes aus dem AVL-Baum.

   c) Falls ein vertikales Segment (a_i) gefunden wird, schaue im AVL-Baum nach, welche y-Koordinaten im y-Intervall des vertikalen Segmentes liegen.

Die Laufzeit ist auch hier $O(n \log_2 n)$, da sich nie mehr als n Objekte im AVL-Baum befinden können. Denn jedes Objekt kann ja nur einmal anfangen.

Also kann in Zeit $O(n \log_2 n)$ entschieden werden, ob sich n horizontale und n vertikale Liniensegmente schneiden.