# Informatik II – Datenstrukturen und Algorithmen in C/C++

Dr. Stefan Lewandowski*

Version 26. April 2011



## Inhaltsverzeichnis



*E-Mail: slewand@slewand.de

# 0 Einführung

## 0.1 Ein kurzer Überblick

Im zweiten Semester beschäftigen wir uns ausführlich mit vielen Standard-Datenstrukturen und typischen Algorithmen, die in der Praxis breite Anwendung finden. Die Schwerpunkte liegen dabei auf Such- und Sortieralgorithmen sowie den wichtigsten Algorithmen auf Graphen – die Kenntnis dieser Standard-Datenstrukturen und Algorithmen ist das erste große Lernziel des zweiten Semesters.

Dabei werden wir feststellen, dass verschiedene Techniken im Entwurf der Algorithmen verwendet werden (und meist auch die Beweismethodik zur Korrektheit der Algorithmen mit der Entwurfsmethodik korreliert). Diese Methoden lassen sich auf viele Problemstellungen übertragen. Jenachdem, welche Struktur ein Problem mit sich bringt, eignet sich die eine oder andere Entwurfsmethode besser – dies in der Praxis einschätzen zu können und so zielgerichtet effiziente Lösungen zu finden, ist das zweite große Lernziel.

Gerade weil viele Wege zu lauffähigen Programmen führen, müssen wir Algorithmen bewerten und deren Effizienz vergleichen lernen – neben dem Speicherverbrauch spielt insbesondere die Laufzeit für umfangreiche Probleme eine entscheidende Rolle. Um sich in der Praxis nicht nur auf einfache Laufzeittests verlassen zu müssen, die stets nur ein unvollständiges Abbild liefern können, sollte man einen Grundstock an theoretischen Aufwandsabschätzungen beherrschen. Darin besteht das dritte große Lernziel.

**Datenstrukturen & Algorithmen, Entwurfsmethoden, Aufwandsabschätzungen**

Wir haben hier die Wahl, nach welchem Kriterium wir die Vorlesung aufbauen wollen:

- nach Anwendungsgebiet – z. B. zuerst alle Suchalgorithmen, dann alle Sortieralgorithmen und dann alle Graphalgorithmen
- nach Entwurfsmethodik – z. B. zuerst alle Backtracking-Methoden, dann alle Greedy-Verfahren, dann alle Teile-und-Herrsche-Verfahren und schließlich die Algorithmen, die Dynamisches Programmieren verwenden
- nach (Laufzeit-)Komplexität – z. B. erst die schnellen Verfahren, dann die langsameren

Die letzte Variante macht wenig Sinn – von den ersten beiden entscheiden wir uns hier für die nach Anwendungsgebiet. Somit werden zwischendurch immer wieder Querverweise bzgl. gleicher Entwurfsmethodiken erfolgen.

## 0.2 Materialien und Literatur

Zu Datenstrukturen und Algorithmen gibt es sehr viel Literatur, ich möchte hier nur ein „Lesebuch“ und zwei Standardlehrbücher vorstellen, sowie auf ein weiteres Skript verweisen, das inhaltlich die hier behandelten Themen ausführlich darstellt und einen guten Ausblick auf verwandte Datenstrukturen und Algorithmen gibt. Sie sollten sich bei der Auswahl der Literatur nicht durch die Programmiersprache C/C++ beschränken lassen – die Anpassung der Algorithmen von dem meist verwendeten Pseudo-Code ist in der Regel leicht möglich.

- Ein schönes Buch als Einstieg ist das „Taschenbuch der Algorithmen“ (B. Vöcking u. a. (Hrsg.)), eine gelungene Sammlung nicht nur trocken mathematisch formulierter Algorithmen, die trotzdem fundiert vorgestellt werden. Neben vielen Aspekten zu Such- und Sortieralgorithmen bekommt man einen guten Einblick in weitere Themengebiete der Informatik. Außerdem ist das Buch mit 20 Euro recht preiswert. Die Algorithmen sind auch online unter `http://www.informatikjahr.de/algorithmus` zu finden.

- Die „Bibel“ – Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest, Clifford Stein: Introduction to Algorithms – deckt nicht nur die hier behandelten Themen ab, sondern bietet auf weit über 1000 Seiten einen sehr umfassenden Einblick in die weite Welt der Informatik. Wer sich nur ein Buch zu Datenstrukturen und Algorithmen kaufen will, ist hier richtig. Inzwischen gibt es unter dem Titel „Algorithmen – eine Einführung“ auch eine deutsche Ausgabe.[1]

- Im deutschsprachigen Raum wird auch häufig auf das Buch von Thomas Ottmann und Peter Widmayer: Algorithmen und Datenstrukturen zurückgegriffen.

- Von den im Netz zur Verfügung stehenden Skripten möchte ich hier nur das von Prof. Claus zur Einführung in die Informatik 2 empfehlen: `http://tinyurl.com/unistgt-inf2skript-ss09`.[2]

Auch, wenn in dem hier vorliegenden Skript zur Vorlesung immer mal wieder auf Wikipedia-Artikel verwiesen wird, sei ausdrücklich davor gewarnt, diese kritiklos für 100%-ig richtig zu halten. Leider ist die Qualität der Artikel bzgl. der Informatik in der Regel eher mäßig.

In diesem Skript gibt es zwei Arten von Randbemerkungen, die die Navigation etwas erleichtern sollen:

- Da neben den alten Klausuraufgaben (am Ende des Skripts) auch viele Übungsaufgaben im Skript verteilt sind, wird auf diese durch eine entsprechende Randbemerkung aufmerksam gemacht. Übung

- Ebenso gibt es kursiv gesetzte Randbemerkungen – diese stellen jeweils das erste Auftreten eines wichtigen Begriffes oder die Position, wo dieser definiert wird, dar. *wichtiger Begriff*

## 0.3 Aufwandsabschätzungen – die $O$-Notation (Landau-Symbole)

Die Entwicklung der Computer ist rasant: Seit Jahrzehnten verdoppelt sich die Anzahl der Transistoren auf den Prozessoren der handelsüblichen PCs ziemlich genau alle 24 Monate, die Rechengeschwindigkeit und der zur Verfügung stehende Speicherplatz erhöhen sich in ähnlicher exponentieller Geschwindigkeit.

Bisher haben wir den Aufwand eines Algorithmus nur am Rande betrachtet. Der Speicherverbrauch ist heutzutage – im Vergleich zur Situation bis vor wenigen Jahren – nur noch von geringerer Bedeutung. Die Laufzeiten bleiben aber weiterhin wichtig. Dabei spielen für die Praxis konstante Faktoren nur eine untergeordnete Rolle (diese werden mit der jeweils nächsten oder übernächsten Rechnergeneration ausgeglichen).

[1] Es soll hier nicht unerwähnt bleiben, dass die hier behandelten AVL-Bäume im Buch leider nur unzureichend kurz erwähnt werden.

[2] vollständige URL: `http://www.ipvs.uni-stuttgart.de/abteilungen/bv/lehre/lehrveranstaltungen/vorlesungen/SS09/Informatik_II_material/start/SkriptMaterial`

Die hauptsächliche Motivation zur $O$-Notation ist die Bewertung und der Vergleich der Effizienz von Algorithmen. Die Aufwandsabschätzungen werden in Abhängigkeit der Problemgröße angegeben – dies ist i.A. die Anzahl der zu bearbeitenden Elemente oder z. B. die Länge der eingegebenen Zeichenfolge. Wir konzentrieren uns dabei auf große Problemgrößen – es interessiert also nur das asymptotische Laufzeitverhalten. Dieses wird formal in der $O$-Notation erfasst.

Zur Verdeutlichung des Einflusses konstanter und nicht-konstanter Faktoren auf die Problemgrößen, die in einer gewissen Zeit bearbeitet werden können, betrachten wir als Beispiel einen Prozessor A, der $10^8$ Schritte pro Sekunde ausführen kann, und einen Prozessor B, der $2 \cdot 10^8$ Schritte pro Sekunde ausführen kann (die Angaben für Prozessor B gelten gleichermaßen für Prozessor A, wenn dieser zwei Stunden Zeit hat). Hat das zu lösende Problem den Aufwand $f(n)$, so lassen sich folgende Problemgrößen in einer Stunde lösen – in der letzten Spalte ist angegeben, wie sich die Laufzeit (bei gleichem Prozessor) verändert, wenn sich die Problemgröße verdoppelt[3].

| Aufwand $f(n)$ des Problems | In 1 Stunde lösbare Problemgröße mit Prozessor A | mit Prozessor B | Laufzeitveränderung bei doppelter Problemgröße |
|---|---|---|---|
| $\log(n)$ | $2^{3.6 \cdot 10^{10}}$ | $2^{7.2 \cdot 10^{10}}$ | +1 Schritt |
| $2 \cdot \log(n)$, $\log(n^2)$ | $2^{1.8 \cdot 10^{10}}$ | $2^{3.6 \cdot 10^{10}}$ | +2 Schritte |
| $n$ | $3.6 \cdot 10^{10}$ | $7.2 \cdot 10^{10}$ | Faktor 2 |
| $2 \cdot n$ | $1.8 \cdot 10^{10}$ | $3.6 \cdot 10^{10}$ | Faktor 2 |
| $n \cdot \log(n)$ | $\approx 1.2 \cdot 10^9$ | $\approx 2.3 \cdot 10^9$ | etwas mehr als Faktor 2 |
| $2 \cdot n \cdot \log(n)$ | $\approx 6.2 \cdot 10^8$ | $\approx 1.2 \cdot 10^9$ | etwas mehr als Faktor 2 |
| $n^2$ | $\approx 1.9 \cdot 10^5$ | $\approx 2.7 \cdot 10^5$ | Faktor 4 |
| $2 \cdot n^2$ | $\approx 1.3 \cdot 10^5$ | $\approx 1.9 \cdot 10^5$ | Faktor 4 |
| $n^3$ | $\approx 3300$ | $\approx 4160$ | Faktor 8 |
| $2 \cdot n^3$ | $\approx 2620$ | $\approx 3300$ | Faktor 8 |
| $2^n$ | $\approx 35$ | $\approx 36$ | Aufwand quadriert sich |
| $3^n$ | $\approx 22$ | $\approx 22$ | Aufwand quadriert sich |

Bzgl. der $O$-Notation unterscheiden wir die Effizienz der Algorithmen, die sich in obiger Tabelle nur um den Faktor 2 unterscheiden, nicht ($2^n$ und $3^n$ sind bzgl. $O$-Notation verschieden!). Wir sehen auch, dass uns Laufzeitgewinne von einem konstanten Faktor nicht helfen, wirklich signifikant größere Probleme zu lösen. Dies wird mit der $O$-Notation abgebildet.

*O-Notation*

Die $O$-Notation, das „$O$“ steht für Ordnung, definiert die Umschreibung „höchstens um einen konstanten Faktor größer (und nur endliche viele Ausnahmen)“ formal. Es gilt für Funktionen $f, g : \mathbb{N} \to \mathbb{N}$:

- $f \in O(g) \;:\Leftrightarrow\; \exists c \in \mathbb{R}^+,\ n_0 \in \mathbb{N} :\ f(n) \leq c \cdot g(n),\ n \geq n_0$,
  d. h., $f$ wächst asymptotisch nicht schneller als $g$ – wir können hier auch die Eselsbrücke „$O$“ wie obere Schranke (für die Laufzeit) benutzen

[3] Dies ist unabhängig davon, ob mit Prozessor A oder B gearbeitet wird!

Neben dem Groß-„$O$“ gibt es noch weitere Symbole, die wir hier zwar nicht näher betrachten, der Vollständigkeit halber aber trotzdem nennen:

- $f \in o(g)$ – $f$ wächst asymptotisch echt langsamer als $g$
- $f \in \Omega(g)$ – $f$ wächst asymptotisch nicht langsamer als $g$
- $f \in \omega(g)$ – $f$ wächst asymptotisch echt schneller als $g$
- $f \in \Theta(g)$ – $f$ wächst asymptotisch gleich schnell wie $g$

Wenn wir für einen Algorithmus die Laufzeit in $O$-Notation angeben wollen, werden wir stets versuchen, eine möglichst langsam wachsende Funktion zu finden, die den Aufwand des Algorithmus im schlechtesten Fall beschreibt[4] (eben eine möglichst gute obere Schranke für die Laufzeit).

Untersuchen wir als Beispiel das Laufzeitverhalten des Sortierens durch Minimumsuche (das Verfahren wurde im ersten Semester bereits vorgestellt – zur Erinnerung: suche das kleinste Element und tausche es an die erste Position, suche jeweils im verbleibenden Feld das kleinste Element und tausche es an die jeweils nächste Position), so werden wir unabhängig von den konkreten Zahlen im zu sortierenden Array stets etwa $n^2/2$ Vergleiche benötigen und etwa ebenso viele Schritte in den Schleifen. Variabel ist hier lediglich, wie oft wir bei der Suche nach dem Index des kleinsten Elements ein kleineres gefunden haben und uns nun den neuen Index merken. Die Gesamtzahl der Schritte wird also etwa irgendwo zwischen 2 und 2.5 mal $n^2$ liegen (je nachdem, was wir als Schritt auffassen, können die Faktoren auch etwas höher sein).

Wir haben also in jedem Ablauf einen Aufwand, der proportional zu $n^2$ ist. In $O$-Notation sagen wir: der Algorithmus hat den Aufwand $O(n^2)$ – der exakte Faktor interessiert uns hier nicht (und entbindet uns damit auch von der Notwendigkeit genau zu überlegen, was als Schritt zu zählen ist[5]).

### 0.3.1 Programmkonstrukte und deren Laufzeiten

Um für ein Programm oder ein Programmfragment die Laufzeit zu bestimmen, muss man den Aufwand für die einzelnen Anweisungen zusammenaddieren.

Als Grobregel kann man sagen: Suche die Stelle im Programm, an der am meisten Schleifen ineinander geschachtelt sind. Ist $k$ diese Anzahl und ist die Anzahl der Schleifendurchläufe jeweils in etwa $n$ (die Problemgröße), so beträgt der Aufwand des Programms $O(n^k)$. Dies ist natürlich nur eine sehr oberflächliche Betrachtung, die im konkreten Fall auch falsch sein kann: nicht jede Schleife hat $\approx n$ Durchläufe und bei Verwendung von Funktionsaufrufen innerhalb von Schleifen ist der Aufwand pro Schleifendurchlauf in der Regel nicht konstant.

[4] Man könnte auch die Laufzeit im Mittel untersuchen, dies ist aber in der Regel deutlich schwieriger. Ohne Zusatzangabe versteht man die $O$-Notation stets als Aufwand im schlechtesten Fall.

[5] Man kann die Anweisung a=b+c; als 1 Schritt auffassen oder als 2 (Addition und Zuweisung) aber auch z. B. als 4 Schritte (lese b aus dem Speicher, lese c aus dem Speicher, führe die Addtion aus, lege den Wert im Speicher als a ab). Dasselbe gilt für Schleifen: entweder 1 Schritt bei jedem Durchlauf oder 2 Schritte (Veränderung der Laufvariablen, Abprüfen der Abbruchbedingung) oder auch mehr (wenn z. B. die Abbruchbedingung in mehrere Schritte aufgeteilt wird).

Mit Hilfe folgender Übersicht kann man den Aufwand gewissenhafter und genauer abschätzen:

- Folge von Anweisungen (oder Blöcken): der Gesamtaufwand ist die Summe der Einzelaufwände (bei Funktionsaufrufen, kann dieser von den Parametern abhängig sein).
- **for**-Schleifen (bedingt auch **while**-Schleifen): Anzahl der Durchläufe mal Aufwand für den Schleifenrumpf – ist der Aufwand abhängig von der Schleifenvariable, so können wir entweder uns mit einer groben Abschätzung begnügen (indem wir für den Aufwand für den Schleifenrumpf das Maximum der möglichen Fälle verwenden) oder die genaue Summe über die Einzelaufwände betrachten.
- Fallunterscheidung: hier betrachten wir die Summe(!) der Einzelaufwände (eigentlich reicht das Maximum – die Summe ist jedoch höchstens doppelt so groß wie das Maximum, der konstante Faktor verschwindet in der $O$-Notation wieder und mit der Summe sind die Aufwandsfunktionen meist einfacher als wenn die Maximumbildung erhalten bleibt).
- Rekursion: führt auf Gleichungen für den Aufwand – hier hilft erstmal nur Übung um zu erkennen, welcher Aufwand letztlich nötig ist. Hat man eine Idee, was herauskommen könnte, so kann man versuchen, die Vermutung mit Hilfe der Gleichungen und vollständiger Induktion zu beweisen.

Führt man die Analyse so durch, erhält man – zumindest in Programmen ohne Rekursion – oft ein Polynom der Form $a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_0$, für die Angabe in $O$-Notation ist dann nur das $n^k$ entscheidend (die anderen Summanden wachsen ja mit mindestens dem Faktor $n$ langsamer, diese werden für größere $n$ also keine Rolle spielen) und wir geben als Aufwand $O(n^k)$ an (ohne den Faktor $a$).

Betrachten wir als Beispiel mit Rekursion das im ersten Semester vorgestellte Beispiel der Türme von Hanoi: Ist nur 1 Scheibe zu bewegen, beträgt der Aufwand $t(1) = 1$. Um den Turm der Höhe $h$ zu bewegen, müssen zweimal ein Turm der Höhe $h-1$ und eine einzelne Scheibe bewegt werden, also $t(h) = 2 \cdot t(h-1)+1$. Die Vermutung $t(h) = 2^h - 1$ lässt sich mit Induktion leicht zeigen: $t(1) = 2^1 - 1 = 1$ und $t(h) = 2 \cdot t(h-1)+1 = 2 \cdot (2^{h-1}-1)+1 = 2^h - 1$. Als Aufwand wird somit $O(2^h)$ angegeben. Weitere allgemeinere Hinweise zur Lösung solcher Rekursionsgleichungen liefert das Master-Theorem (`http://de.wikipedia.org/wiki/Master-Theorem`).

Manchmal lassen sich auch bessere Abschätzungen als über obiges Standardvorgehen erreichen – wir werden dies z. B. beim Dijkstra-Algorithmus zur Bestimmung kürzester Wege sehen.

### 0.3.2 Arbeiten mit der $O$-Notation

Vergleicht man zwei Algorithmen miteinander, so erhält man zunächst zwei Abschätzungen in $O$-Notation. Oft sieht man diesen sofort an, welche Funktion langsamer wächst: $n^k$ wächst langsamer als $n^{k'}$ genau dann, wenn $k < k'$ gilt; $n \cdot \log(n)$ wächst langsamer als $n^2$. Bei anderen Funktionen ist es u.U. nicht gleich klar: wächst $n^k$ für jedes $k$ langsamer als $2^n$; wächst $n \cdot \log^k(n)$ langsamer als $n\sqrt{n}$ (und wenn ja, gilt dies für jedes $k$)?

Formal müssten wir jeweils Konstanten $c$ und $n_0$ finden, so dass die Ungleichung in der Definition erfüllt ist. In der Praxis hilft uns die Grenzwertbestimmung aus der Mathematik weiter und wir können uns oft folgender Technik bedienen:

Für fast alle Fälle kann man über $\lim_{n\to\infty}(f(n)/g(n))$ bestimmen, in welchem Verhältnis bzgl. $O$-Notation die beiden Funktionen $f$ und $g$ zueinander stehen.

- Bei $\lim \to 0$ wächst $f$ echt langsamer (also $f \in O(g)$),
- Divergiert der Quotient ($\lim \to \infty$), so wächst $f$ echt schneller (also $f \notin O(g)$ und $g \in O(f)$, da $\lim_{n\to\infty}(g(n)/f(n)) \to 0$),
- Gilt $\lim \to c$ für eine Konstante $c$ ungleich 0, so unterscheiden sich die beiden Funktionen ab einem $n_0$ nur noch maximal um einen konstanten Faktor, somit $f \in O(g)$ und zusätzlich auch $g \in O(f)$.

Existiert der Grenzwert nicht, dann sind die Funktionen entweder bzgl. der $O$-Notation nicht vergleichbar (in der Praxis selten – es gilt dann $f \notin O(g)$ und $g \notin O(f)$) oder der Quotient bewegt sich in einem beschränkten Bereich, ist aber nicht konvergent (ebenfalls selten – es gilt dann $f \in O(g)$ und $g \in O(f)$), Beispiele selbst überlegen.

**Warum stimmt das?** Die mathematische Definition eines Grenzwerts liefert die Antwort: $\lim_{n\to\infty}(f(n)/g(n)) = c_l$ heißt, dass es für jedes $\varepsilon > 0$ ein $n_0 \in \mathbb{N}$ gibt, so dass sich der Quotient $f(n)/g(n)$ für $n > n_0$ von $c_l$ nicht mehr als $\varepsilon$ unterscheidet. Ist obiger Grenzwert z. B. 3, wissen wir dass ab einem bestimmten $n_0$ der Quotient $f(n)/g(n) \leq 4$ ist. Dieses $n_0$ und die Wahl $c = c_l + \varepsilon$ erfüllen dann die geforderte Eigenschaft in der Definition der $O$-Notation. Das Wissen um die Existenz dieser Werte reicht dabei aus, wir müssen die konkreten Werte nicht bestimmen.

**Der Einfluss der (vernachlässigten) Konstanten:** Sind die Funktionen bzgl. $O$-Notation gleich, so kann man sich die Algorithmen genauer anschauen, ob sich die Konstanten stark voneinander unterscheiden – meist ist dies nicht der Fall, sodass man den leichter zu programmierenden Algorithmus nimmt oder den, der sich flexibler an die konkrete Anwendung anpassen lässt. Sind die Funktionen verschieden, so ist der nichtkonstante Faktor meist ab relativ kleinem $n$ deutlich größer als der Unterschied zwischen den Konstanten. Einzig, wenn der Unterschied nur ein logarithmischer Faktor ist, sollte die Größe der Konstanten genauer untersucht werden: ist der Faktor z. B. bei einem $O(n)$-Algorithmus 50 mal größer als beim alternativ zu verwendenden $O(n \cdot \log(n))$-Algorithmus, so müsste $\log(n) > 50$ werden (also $n > 2^{50}$), damit sich hier ein Geschwindigkeitsvorteil ergibt. Solche großen Unterschiede sind in den Konstanten aber extrem selten.

**Übungsbeispiele:** Zeigen Sie, dass $(n + a)^b \in O(n^b)$ für beliebige $a, b \in \mathbb{N}$. Übung

Für die Grenzwertbestimmung leisten die Hilfsmittel der Mathematik, speziell die Regel von L'Hospital, gute Dienste. Benutzen Sie diese, um zu zeigen, dass $n \cdot \log^k(n) \in O(n\sqrt{n})$ (dies gilt für beliebiges $k \in \mathbb{N}$ – beginnen Sie mit $k = 1$ und verallgemeinern Sie das Vorgehen auf größere Werte für $k$).

# 1 Suchalgorithmen

Das Wesen der Suche ist in einer Menge von Elementen ein (oder mehrere) Element(e) mit bestimmten Eigenschaften zu finden.

## 1.1 Die Problemlösestrategien Backtracking (erschöpfende Suche), Dynamisches Programmieren und gierige Verfahren

**Ein Beispiel-Problem: die gerechte Erbschaft:** Gegeben ist ein Tupel $M$ aus Werten $M = (g_1, g_2, \ldots, g_k)$, gesucht ist eine Partition in zwei Teile, so dass die Summe der Werte in beiden Teilen gleich ist, falls solch eine Aufteilung möglich ist, oder eine Mitteilung, dass solch eine Aufteilung nicht möglich ist.

Betrachten wir folgende Probleminstanz mit 12 Werten (Gesamtsumme 300):

72 49 42 26 23 22 19 19 15 8 4 1

Erste Idee: Sortiere die Werte absteigend (ist hier schon geschehen), füge den jeweils nächsten Wert zu der zu dem Zeitpunkt kleineren Teilmenge (bzgl. der Summe der Werte). Dies führt zu der Aufteilung:

- 72, 26, 22, 19, 8, 4 (Summe 151)
- 49, 42, 23, 19, 15, 1 (Summe 149)

*Heuristik*

Solche Heuristiken (dies sind Algorithmen, die schnell gute Näherungslösungen finden) sind in der Informatik verbreitet – speziell für Probleme, deren exakte Lösung nur mit großem Aufwand zu finden ist.

*Backtracking*

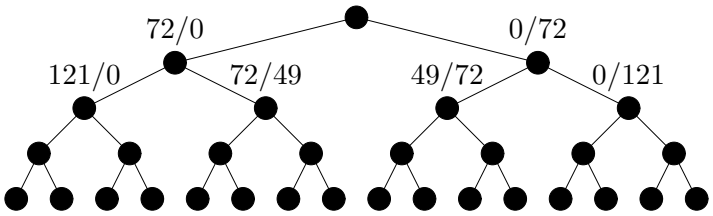
Für unser konkretes Problem hilft uns diese Näherungslösung nur bedingt weiter – wir wissen damit nicht, ob es nicht vielleicht doch eine andere Möglichkeit gibt, mit der sich die Elemente in zwei gleichwertige Mengen aufteilen lassen. Wenn einem nichts besseres einfällt, muss man alle Möglichkeiten systematisch durchtesten. Stellt sich dabei ein eingeschlagener Weg als Sackgasse heraus, macht man den zuletzt durchgeführten Schritt rückgängig und versucht eine andere Möglichkeit – dieses Vorgehensprinzip nennt man Backtracking. Da das Verfahren in Theorie und Praxis meist sehr aufwendig ist, wird es nur dann eingesetzt, wenn sich keine Strukturen im Problem zur effizienteren Lösungsfindung verwenden lassen.

**Ein exaktes Verfahren:** Wenn es eine Lösung gibt, dann gehört das erste Element entweder zur ersten Teilmenge oder zur zweiten. Da wir es vorab nicht wissen, müssen wir beide Möglichkeiten testen. Die restlichen Elemente sind dann wiederum jeweils einer der beiden Teilmengen zuzuweisen. Sind alle Elemente verteilt und ist die Summe der Elemente der ersten Teilmenge gleich der der zweiten, gib die Lösung aus. Diese umgangssprachliche rekursive Formulierung lässt sich leicht in Pseudocode beschreiben:

```
Prozedur verteile(Element,SummeErbeEins,SummeErbeZwei) {
  wenn alle Elemente verteilt sind und SummeErbeEins=SummeErbeZwei
    gib die Lösung aus
  sonst
    wenn noch Elemente zu verteilen sind
      füge das Element zur ersten Teilmenge hinzu
      verteile(nächstes Element,SummeErbeEins+Wert(Element),SummeErbeZwei)
      füge das Element stattdessen zur zweiten Teilmenge hinzu
      verteile(nächstes Element,SummeErbeEins,SummeErbeZwei+Wert(Element))
}
```

Die Elemente werden also der Reihe nach entweder in die eine oder andere Teilmenge eingefügt. Ist eine Verteilung als Sackgasse erkannt worden, so macht man den letzten Schritt rückgängig und versucht ggf. das Element statt in die erste in die zweite Teilmenge einzufügen. Die verschiedenen Aufteilungsmöglichkeiten lassen sich als Entscheidungsbaum darstellen:

*Entscheidungsbaum*




verteile 72 auf Erbe 1 oder 2
verteile 49 auf Erbe 1 oder 2
verteile 42 auf Erbe 1 oder 2
...

Eine gerechte Aufteilung des Erbes gibt es genau dann, wenn in der untersten Ebene wenigstens ein Knoten mit Beschriftung 150/150 dabei ist – der Weg von der Wurzel zu diesem Knoten gibt an, wie die Elemente auf die beiden Hälften zu verteilen sind. Da man sich im Entscheidungsbaum stets zunächst so weit wie möglich in die tieferen Ebenen begibt (und bei Sackgassen wieder aufsteigt), spricht man beim Backtracking auch von Tiefensuche.

*Tiefensuche*

Zur Umsetzung in C/C++ fehlen hier natürlich noch ein paar Details, wir geben hier nur den Code an. Vollziehen Sie diesen nach, führen Sie diesen an ein paar Beispielen von Hand oder programmgestützt aus und machen sich den Ablauf klar.

Übung

gerechte_erbschaft.h

```
#ifndef GERECHTE_ERBSCHAFT_H
#define GERECHTE_ERBSCHAFT_H

void gerechte_erbschaft (const int [], const int ); // Array + dessen Größe

#endif
```

gerechte_erbschaft.cpp

```
#include "gerechte_erbschaft.h"
#include <iostream>
using namespace std;

// nur modulweit bekannte Variablen

const int* erbe; // eigentlich ein Array, aber wir kennen die Größe
// a priori nicht; der Zugriff *erbe ist gleichbedeutend mit erbe[0]
// ein Array ist intern immer ein Pointer auf den ersten Eintrag
bool* inErbeEins; // dito, ein Array, das anzeigt, welche Werte im Erbe 1 sind
int size;
int gesamterbe; // für eine später eingeführte Variante

void verteile_erbe(const int erbteil,
                   const int summeErbeEins, const int summeErbeZwei);
// Vorwärtsdeklaration der lokalen Funktion
```

```
void gerechte_erbschaft(const int erbe_pa[], const int size_pa) {
  // erbe_pa : Zeiger auf der erste Element des Arrays der Erbteile
  // size_pa : Anzahl der Elemente
  erbe=erbe_pa; // für die anderen Funktionen im Modul sichtbar machen
  size=size_pa; // für die anderen Funktionen im Modul sichtbar machen
  inErbeEins=new bool[size_pa]; // Initialisierung hier nicht nötig
  verteile_erbe(0,0,0);
  delete inErbeEins; // Speicher wieder freigeben
}

void verteile_erbe(const int erbteil,
                   const int summeErbeEins, const int summeErbeZwei) {
  if (erbteil==size) { // das letzte Erbteil ist bereits verteilt
    if (summeErbeEins==summeErbeZwei) {
      cout << "Neue Loesung:" << endl << "Erstes Erbe: ";
      for(int i=0;i<size;i++) {
        if (inErbeEins[i]) {
          cout << erbe[i] << ", ";
        }
      }
      cout << "zweites Erbe: ";
      for(int i=0;i<size;i++) {
        if (!inErbeEins[i]) {
          cout << erbe[i] << ", ";
        }
      }
      cout << "suche weitere Loesungen ..." << endl;
    } // else alle Elemente verteilt, aber keine gerechte Aufteilung
  } else {
    inErbeEins[erbteil]=true;
    verteile_erbe(erbteil+1,summeErbeEins+erbe[erbteil],summeErbeZwei);
    inErbeEins[erbteil]=false;
    verteile_erbe(erbteil+1,summeErbeEins,summeErbeZwei+erbe[erbteil]);
  }
}
```

Im Hauptprogramm hätte man z. B. ein vordefiniertes (oder vom Benutzer eingelesenes) Array **int** erbschaft[12] = {72,49,42,26,23,22,19,19,15,8,4,1}; und würde die möglichen Lösungen mithilfe des Aufrufs gerechte_erbschaft(erbschaft,12); berechnen.

**Wie groß ist der Aufwand?** Offensichtlich existieren $2^n$ Möglichkeiten, die Elemente den zwei Erbteilen zuzuweisen (2 Möglichkeiten für das erste Element, für jede dieser 2 Möglichkeiten je 2 für das zweite Element, für jede dieser 4 Möglichkeiten je 2 für das dritte usw.). Die Ausgabe erfordert $O(n)$ Schritte. Der Gesamtaufwand zum Finden und Ausgeben aller Lösungen beträgt damit $O(n \cdot 2^n)$ (der lineare Aufwand zur Ausgabe fällt nur bei einer gefundenen Lösung an – überlegen Sie sich, ob der Aufwand nicht vielleicht etwas geringer ist: der Aufwand zum Ablaufen des Entscheidungsbaums beträgt $O(2^n)$, gäbe es z. B. höchstens $O(n^k)$ mögliche Lösungen, dann wäre der Gesamtaufwand $O(2^n + n \cdot n^k) = O(2^n)$). Interessiert nur die erste gefundene Lösung, so beträgt der Aufwand $O(2^n + n) = O(2^n)$. Übung

**Beschleunigung:** Bei genauerer Betrachtung lassen sich einige Sachen hier effizienter gestalten:

- Ist die Summe der Elemente ungerade, kann es keine Lösung geben.
- Das erste Element lässt sich fest dem ersten Erbe zuweisen – zu jeder Lösung gibt es eine symmetrische, die Erbe 1 und 2 komplett vertauscht.
- Wir brauchen nur das eine Erbe explizit zu verwalten und dessen Summe als Parameter in der Rekursion mitzugeben – das zweite Erbe ergibt sich automatisch.
- Frühzeitiges Erkennen von Lösungen: hat das erste Erbe den Zielwert erreicht, müssen alle weiteren Elemente dem zweiten Erbe zugewiesen werden.
- Frühzeitiges Erkennen von Sackgassen
  - wenn das eine Erbe die Hälfte überschritten hat, kann keine gerechte Erbschaft mehr entstehen, der entsprechende Zweig im Entscheidungsbaum braucht nicht weiter verfolgt zu werden
  - dies gilt ebenso, wenn die verbleibenden Elemente nicht mehr ausreichen, um das erste Erbe auf die Hälfte aufzufüllen

*Branch-and-Bound*

Die Umsetzung dieser Beobachtungen führt dazu, dass wir Teile des Entscheidungsbaums nicht mehr betrachten – die Technik, auf diese Art und Weise Backtracking-Verfahren zu beschleunigen, nennt man Branch-and-Bound.

Übung

Wir überlassen die Umsetzung der Einsparmöglichkeiten und deren experimentelle Überprüfung als Programmierübung.

*Dynamische Programmierung*

**Ein weiterer Ansatz zur Lösung** führt eine neue Entwurfsmethodik ein: die Dynamische Programmierung[6]. In obigem rekursiven Backtracking-Ansatz werden unter Umständen viele gleiche Schritte mehrfach berechnet, es könnte im Entscheidungsbaum auf einem Level mehrere Knoten mit gleicher Beschriftung geben – dieses wird speziell dann auftreten, wenn die Werte der zu verteilenden Elemente relativ klein sind. Wir könnten also versuchen, die Zahlen zu ermitteln, die sich als Summe der einzelnen Werte bilden lassen, wobei uns nur Zahlen interessieren, die kleiner gleich dem Zielwert sind.

- Berücksichtigt man nur das 1. Element, lassen sich die Werte 0 und 72 bilden.
- Berücksichtigt man auch das 2. Element, lassen sich die bisherigen Werte bilden und alle Werte, die sich als Summe der bisherigen Werte und dem 2. Element bilden lassen: neben der 0 und 72 also 49 und 121.
- Berücksichtigt man auch das $i$-te Element, lassen sich die Werte, die die ersten $i-1$ Elemente berücksichtigen, bilden und alle Werte, die sich als Summe der bisherigen Werte und dem $i$-ten Element bilden lassen:
  - für $i = 3$ ergeben sich 0, 49, 72, 121 und zusätzlich 42, 91, 116 (163 ist nicht relevant, da damit die 150 überschritten sind)
  - für $i = 4$ ergeben sich 0, 42, 49, 72, 91, 116, 121 und zusätzlich 26, 68, 75, 98, 117, 142, 147
  - usw.

[6] Der Begriff ist historisch gewachsen – für die Praxis bedeutet es Vermeidung von Mehrfachaufwand durch Speicherung von Zwischenergebnissen (in der Regel in Tabellen/Arrays).

Wir bestimmen also eine Lösung eines größeren Problems unter Verwendung der Lösungen kleinerer Teilprobleme (kleiner im Sinne von weniger berücksichtigten Elementen – bei der gerechten Erbschaft gibt es nur das eine kleinere Teilproblem, bei anderen Problemstellungen können dies aber auch mehrere sein) und einem Auswahlprozess (hier: das neue Element gehört zum ersten Erbe oder eben nicht).

Für die Umsetzung in C/C++ wählen wir hier einen sehr einfachen Ansatz: wir markieren uns in einem **bool**-Array (Indizes 0 bis Zielwert) die Werte, die erreicht werden können. Ist am Ende der Zielwert markiert, so gibt es eine Lösung.

`gerechte_erbschaft.cpp` (Fortsetzung)

```
void gerechte_erbschaft_dynprog(const int erbe_pa[], const int size_pa) {
  // erbe_pa : Zeiger auf der erste Element des Arrays der Erbteile
  // size_pa : Anzahl der Elemente
  gesamterbe=0;
  int i,j;
  for(i=0;i<size_pa;i++) {
    gesamterbe+=erbe_pa[i];
  }
  if (gesamterbe%2==0) { // was passiert, wenn man die Abfrage weglässt?
    int zielwert=gesamterbe/2;
    bool* erreichbar=new bool[zielwert+1];
    erreichbar[0]=true;
    for(i=1;i<=zielwert;i++) {
      erreichbar[i]=false;
    }
    for(j=0;j<size_pa;j++) {
      for(i=zielwert-erbe_pa[j];i>=0;i--) { // rückwärts, damit Erbteil
        if (erreichbar[i]) {                // nur 1 mal verwendet wird
          erreichbar[i+erbe_pa[j]]=true;
    } } }
    if (erreichbar[zielwert]) {
      cout << "Es gibt ein gerechtes Erbe" << endl;
    } else {
      cout << "Es gibt kein gerechtes Erbe" << endl;
    }
    delete erreichbar; // Speicher wieder freigeben
  } else {
    cout << "Es gibt kein gerechtes Erbe" << endl;
  }
}
```

**Wie groß ist der Aufwand mit der Methode Dynamische Programmierung?** Die äußere Schleife hat $n$ Durchläufe, in denen in der inneren Schleife jedesmal das gesamte **bool**-Array durchlaufen wird. Ist $n$ die Anzahl der Elemente und $g$ die Summe der Elemente, so beträgt der Aufwand $O(n \cdot g)$ (der Faktor $\frac{1}{2}$ verschwindet als Konstante im $O$).

Wir sehen: ist das $g$ gegenüber $n$ sehr groß (Größenordnung $> 2^n$), so ist der Backtracking-Ansatz schneller; für größere $n$ (und nicht zu großes $g$) wird man das Ergebnis jedoch nur mit dem Ansatz der Dynamischen Programmierung erleben.

Wir haben mit dem Ansatz der Dynamischen Programmierung nur bestimmt, ob der Zielwert erreicht werden kann, jedoch nicht, wie wir dieses tun! Finden Sie eine einfache Lösung, so Übung

dass der Aufwand $O(n \cdot g)$ erhalten bleibt. (Falls Sie keine Idee haben, können Sie einen Blick in den Abschnitt 3.3 wagen – wir werden da ein ähnlich gelagertes Problem lösen.)

Einen wesentlichen Unterschied gibt es hier noch zwischen den beiden Methoden: Beim Dynamischen Programmieren finden wir nur eine Lösung, beim Backtracking haben wir zumindest die Möglichkeit nach Finden der ersten Lösung auch noch nach weiteren Lösungen suchen zu lassen. Überlegen Sie sich, ob es auch einen Ansatz mit Dynamischer Programmierung gibt, um die Anzahl verschiedener Lösungen zu bestimmen.

*gierige (engl. greedy) Verfahren*

**Eine weitere Problemlösestrategie:** Unser erster Ansatz – das zielgerichtete schrittweise Verteilen der einzelnen Erbstücke auf die zwei Mengen – führte bei der Problemstellung des gerechten Erbes nur auf eine Näherungslösung. Es gibt in der Informatik aber genügend Problemstellungen, bei denen solch ein gieriges Verfahren (engl. greedy) beweisbar auf eine optimale Lösung führt – als klassisches Beispiel werden wir in Abschnitt 3.3 noch den Dijkstra-Algorithmus zur Berechnung kürzester Wege behandeln. Das Wesen solcher Problemstellungen ist, dass wir allein aufgrund der bisherigen Schritte entscheiden können, ob ein als nächster zu wählender Schritt auf eine optimale Lösung führt (lokale Optimalitätseigenschaft) – dies ist bei der gerechten Erbschaft nicht der Fall: ob die Wahl, ein Erbteil dem Erbe 1 zuzuweisen, richtig war, entscheidet sich erst später – bei dem nun folgenden Problem können wir (zumindest in der ersten Variante) immer sofort entscheiden, welches der nächste richtige Schritt sein wird.

Übung ⇝ Labor!

Eine umfangreiche Übungsaufgabe – **das Münzproblem**: Will man ermitteln, wieviel Euro-Cent-Münzen man für einen bestimmten Cent-Betrag benötigt, so kann man dies leicht auf eine gierige Art und Weise bestimmen: Nimm jeweils die größte Münze, so dass die bisherige Summe den Zielbetrag nicht übersteigt. Um z. B. 79 Cent mit möglichst wenig Münzen darzustellen, nimmt man zunächst eine 50-Cent-Münze, dann eine 20-Cent-Münze, eine 5-Cent-Münze und schließlich zwei 2-Cent-Münzen. Mit der üblichen Euro-Aufteilung in die Cent-Werte 50, 20, 10, 5, 2 und 1 führt das gierige Vorgehen stets zur optimalen Lösung. Stellen wir uns für einen Moment jedoch ein anderes Land vor, dass eine – für unsere Verhältnisse – sehr ungewöhnliche Aufteilung in die Cent-Werte 61, 42, 25, 11, 7 und 1 hat, so führt das gierige Verfahren nicht immer zum optimalen Ergebnis (der Wert 21 würde z. B. gierig mit fünf Münzen (11, 7, drei mal 1) dargestellt, wohingegen drei Münzen mit Wert 7 ausreichen). Diese Anfangsüberlegungen führen auf folgende Fragen – entscheiden Sie für jede Teilfrage, ob Sie das Problem mit einem Greedy-Verfahren, dynamischer Programmierung oder Backtracking lösen sollten und programmieren Sie die Funktionen in C/C++ aus:

- Bestimmen Sie für die Beträge von 1 bis 99 Cent jeweils, wieviel Münzen man mindestens braucht, wenn als Unterteilung die Cent-Werte 50, 20, 10, 5, 2 und 1 zur Verfügung stehen, und bilden Sie den Mittelwert. Vergleichen Sie diesen Mittelwert mit der Münzaufteilung, die in anderen Ländern üblich ist (z. B. 50, 25, 10, 5, 2, 1).
- Bestimmen Sie für die Beträge von 1 bis 99 Cent jeweils, wieviel Münzen man mindestens braucht, wenn als Unterteilung die Cent-Werte 61, 42, 25, 11, 7 und 1 zur Verfügung stehen, und bilden Sie den Mittelwert.
- Finden Sie 6 Münzwerte, so dass der damit berechnete Mittelwert der benötigten Münzen für die Beträge 1 bis 99 Cent minimal wird. Welche Lösung findet man, wenn für die 6 Münzwerte zusätzlich gefordert wird, dass die optimale Münzanzahl jeweils gierig erreicht wird?

## 1.2 Suchen in sequentiellen Strukturen

Beim Beispiel der gerechten Erbschaft haben wir eine Lösung in einer Menge gesucht, die wir implizit erst im Ablauf des Algorithmus bestimmt haben: die Menge aller Teilmengen (Potenzmenge). Für alle weiteren Suchalgorithmen gehen wir davon aus, dass wir eine Menge von Daten verwalten und in dieser Menge Elemente suchen wollen.

### 1.2.1 Suchen in unsortierten Listen und Arrays

Liegen die Daten in einer beliebigen Reihenfolge – also völlig unstrukturiert – vor, so bleibt uns auf die Anfrage, ob ein bestimmter Wert in der Liste oder dem Array vorkommt, nur, jedes Element der Liste (oder des Arrays) anzuschauen.[7] Der Aufwand ist $O(n)$.

Schreiben Sie die entsprechenden C/C++-Funktionen – einmal für Arrays, einmal für Listen. Übung

### 1.2.2 Suchen in sortierten Listen und Arrays

Liegen die Daten sortiert in einer Liste, so kann man die Suche frühzeitig abbrechen, der Aufwand bleibt für den schlechtesten Fall (erfolgreiche Suche nach dem größten Element oder erfolglose Suche nach einem noch größeren Wert) bei $O(n)$. Kann man auf jedes Element direkt zugreifen, so gibt es einen Trick, den man schon als Kind bei einfachen Ratespielen kennenlernt: soll man eine Zahl zwischen 0 und 100 erraten und erhält nach jedem Tipp nur die Information, ob die gesuchte Zahl kleiner oder größer ist, so wird man zunächst die 50 raten, um so die Anzahl der möglichen verbleibenden Zahlen möglichst gering zu halten; ist die gesuchte Zahl kleiner, wäre der nächste Tipp 25, dann 37, 43, 40, 41 bis schließlich die 42 gefunden wurde. Bei Arrays wird man natürlich am mittleren Index zuerst nachschauen und dann den nächsten Index entsprechend dem dort gefundenen Wert anpassen. Dieses Verfahren ist unter den Namen Intervallschachtelung und Binäre Suche bekannt.

*Intervallschachtelung, Binäre Suche*

Wir stellen hier zwei Varianten in C/C++ vor, die sich nur in einem kleinen Detail unterscheiden:

`intervallschachtelung.h`

```
#ifndef INTERVALLSCHACHTELUNG_H
#define INTERVALLSCHACHTELUNG_H

bool binsearch1 ( int [], const int , const int , const int ); // Array , gesucht ,...
bool binsearch2 ( int [], const int , const int , const int ); // ..., Index von , bis

#endif
```

[7] Man spricht hier zwar auch von erschöpfender Suche – es gibt dabei aber keine Sackgassen, keine rückgängig zu machenden Schritte, so dass man hier nicht von Backtracking spricht.

intervallschachtelung.cpp

```
#include "intervallschachtelung.h"

bool binsearch1(int a[], const int gesucht, const int von, const int bis) {
  if (bis>=von) {
    int mitte=(von+bis)/2;
    if (a[mitte]==gesucht) {
      return true;
    } else if (a[mitte]>gesucht) {
      return binsearch1(a,gesucht,von,mitte-1);
    } else {
      return binsearch1(a,gesucht,mitte+1,bis);
    }
  } else return false;
}

bool binsearch2(int a[], const int gesucht, const int von, const int bis) {
  if (bis>von) {
    int mitte=(von+bis)/2;
    if (a[mitte]<gesucht) {
      return binsearch2(a,gesucht,mitte+1,bis);
    } else {
      return binsearch2(a,gesucht,von,mitte);
    }
  } else {
    return a[von]==gesucht;
  }
}
```

Die erste Variante ist die naheliegende: Ist das gesuchte Element in der Mitte, so war die Suche erfolgreich, ansonsten wird in der linken oder rechten Hälfte weitergesucht. Die zweite Variante unterscheidet sich insofern, dass die Gleichheit nicht separat abgefragt wird, sondern der entsprechende Index im Intervall verbleibt – erst, wenn das Intervall auf einen Index geschrumpft ist, wird abgefragt, ob dies das gesuchte Element ist. Dieser vermeintliche Nachteil entpuppt sich bei genauerem Hinsehen als Vorteil: bei der ersten Variante sind für jede Intervallhalbierung 2 Vergleiche notwendig, während bei der zweiten Variante nur noch 1 Vergleich dazu aufgewendet wird. Dieser Vorteil wird jedoch mit dem Nachteil erkauft, dass wir keinen „Glückstreffer“ mehr erzielen können – das Intervall wird immer halbiert, bis nur noch ein Element übrig ist.

Übungsaufgaben: Um nicht in der Rekursion immer wieder den Zeiger auf das Array übergeben zu müssen, kann man die Intervallschachtelung auch leicht mit **while**-Schleifen durchführen. Schreiben Sie obige Funktionen entsprechend um. Untersuchen Sie experimentell, welche der obigen Varianten in der Praxis schneller ist, betrachten Sie dabei zum Einen die Fälle, in denen nach einem im Feld vorhandenen Element gesucht wird, und zum Anderen die Fälle, bei denen das Element nicht im Feld vorkommt. Übung

**Aufwandsabschätzung der Intervallschachtelung in $O$-Notation** In beiden Varianten wird das Intervall nach zwei bzw. einem Vergleich halbiert. Ist $n = 2^k$, so ist nach $k$ Halbierungen das Intervall auf die Breite 1 geschrumpft – mathematisch formuliert benötigen wir logarithmisch viele Halbierungen. Die genaue Konstante müssen wir für die $O$-Notation nicht

bestimmen, der Aufwand beträgt für beide Varianten somit $O(\log n)$.[8]

*Interpolationssuche*

**Interpolationssuche:** Sucht man Prof. Weiss im Telefonbuch, so würde man sicher nicht in der Mitte aufschlagen und bei „Lewandowski“ dann feststellen, dass man weiter hinten schauen muss, sondern man hätte gleich fast ganz hinten aufgeschlagen und dann bei „Zschuppan“ nur wenige Seiten zurückgeschlagen. Dieses Prinzip nennt man Interpolationssuche. Voraussetzung dafür ist, dass man mit den Werten im Array rechnen kann. Wird in einem Feld a mit Indizes 11 bis 99, bei dem a[11]=20 und a[99]=284, der Wert 42 gesucht, so gehen wir implizit davon aus, dass die Werte im Feld gleichverteilt sind – die Differenz des letzten und ersten Wertes (284−20) verteilt sich (so die Annahme) gleichmäßig auf die Indizes 11 bis 99, im Beispiel a[11]=20, a[12]=23, ..., a[18]=41, a[19]=44, ..., a[98]=281, a[99]=284 – und damit der gesuchte Wert 42 etwa auf Index 18.333, also Index 18, sein sollte. Allgemein berechnen wir diesen Index mit der Formel

```
von+(gesucht-a[von])*(bis-von)/(a[bis]-a[von])
```

– wir schauen also, welches Intervall die Werte im aktuell noch zu betrachtenden Teilfeld (Indizes von,...,bis) umfasst und interpolieren linear, wo das gesuchte Element in etwa stehen müsste, im Beispiel am Index $11 + \frac{42-20}{284-20} \cdot (99 - 11)$. Die Division erfolgt im C/C++-Code zuletzt, da es sich um eine Ganzzahl-Division handelt (selbst überlegen, was sonst passieren würde).

Die C/C++-Funktion lautet somit:

`intervallschachtelung.cpp` (Fortsetzung)

```
bool interpol1 (int a[], const int gesucht, const int von, const int bis) {
  if (bis>von) {
    int mitte=von+(gesucht-a[von])*(bis-von)/(a[bis]-a[von]);
    if (mitte<von || mitte>bis) { // außerhalb des erlaubten Bereichs
      return false;
    } else if (a[mitte]==gesucht) {
      return true;
    } else if (a[mitte]>gesucht) {
      return interpol1(a,gesucht,von,mitte-1);
    } else {
      return interpol1(a,gesucht,mitte+1,bis);
    }
  } else {
    return a[bis]==gesucht;
    // true ist nur bei Aufruf mit 1 breitem Intervall möglich, sonst war
    // gesucht vorher schon am Rand des Intervalls und wurde dort gefunden
  }
}
```

*Übung*

Die Behandlung der Randfälle ist etwas tückisch, speziell, wenn das gesuchte Element nicht im Feld vorhanden ist. Untersuchen Sie auf zufällig erzeugten Arrays (beachten Sie, dass diese sortiert sein müssen), wieviel Vergleiche im Mittel benötigt werden, wenn das Element im Feld vorhanden ist bzw. wenn das Element nicht vorhanden ist. Untersuchen Sie insbesondere, wie oft der Abbruch dabei in dem letzten **else**-Zweig stattfindet. Hier ist übrigens noch wichtig,

[8] Eine theoretische Untersuchung zum Aufwand der beiden Varianten im Mittel findet sich z. B. unter `http://www.ipvs.uni-stuttgart.de/abteilungen/bv/lehre/lehrveranstaltungen/vorlesungen/WS0809/Informatik_I_material/start/Skriptum/Informatik_I-II_Kap_6-7_1020-1333.pdf` auf den Folien 1126 bis 1138 – vergleichen Sie dieses mit den von Ihnen experimentell ermittelten Werten.

dass a[bis] und nicht a[von] verglichen wird (konstruieren Sie ein Beispiel, bei dem sonst auf einen Index außerhalb des Arrays zugegriffen würde).

Der erwartete Aufwand beträgt lediglich $\log\log n + 1$ Vergleiche (auf die mathematische Herleitung wird hier verzichtet), d. h., selbst bei $n = 2^{32}$ Elementen, hat man im Mittel nach nur 6 Vergleichen das gesuchte Element gefunden. Leider ist dies nur ein Erwartungswert, der gilt, wenn die Daten im Array etwa gleichmäßig auf das zugehörige Intervall verteilt sind (so wie es im Telefonbuch der Fall ist). Im schlimmsten Fall kann der Aufwand deutlich größer als bei der einfacheren binären Suche sein – konstruieren Sie ein Beispiel, dass deutlich mehr Vergleiche als die Binäre Suche benötigt. Übung

Bei der Binären Suche haben wir uns im Mittel Vergleiche sparen können, wenn die Abfrage auf Gleichheit erst erfolgt, wenn die Breite des Intervalls auf 1 geschrumpft ist. Überlegen Sie, ob und wie Sie diese Idee auch auf die Interpolationssuche anwenden können. Übung

## 1.3 Suchbäume

Auf sortierten Arrays ist die Intervallschachtelung für die Praxis ausreichend schnell – das Problem dabei ist, dass wir nur mit größerem Aufwand neue Elemente einfügen und löschen können – der notwendige lineare Aufwand ist zu hoch, wenn relativ viele Einfüge- und Lösch-Operationen nötig sind und nur relativ wenig Such-Operationen. Bisher haben wir sich dynamisch ändernde Daten mit Listen verwaltet. Beim Einfügen und Löschen sind hier zwar etwas weniger Operationen notwendig, der für die Intervallschachtelung benötigte wahlfreie direkte Zugriff auf jedes Element fehlt jedoch. Suchbäume sollen hier helfen, Daten effizient zu verwalten. Das Ziel ist, nicht nur die Suche in $O(\log n)$ Schritten durchzuführen, sondern auch das Einfügen und Löschen!

### 1.3.1 Bäume und Binärbaume

Baumartige Strukturen sind in der Informatik häufig, entsprechend differenziert sind die verwendeten Begriffe. Eine Variante von Bäumen – den Entscheidungsbaum – haben wir schon kennengelernt. Ein Baum setzt sich zusammen aus Knoten, die mit Kanten verbunden sind. Die Beziehungen zwischen den Knoten beschreibt man analog zu familiären Beziehungen: Der Elter-Knoten hat einen oder mehrere Kinder-Knoten (untereinander Geschwisterknoten genannt)[9], analog spricht man von Vorgänger- und Nachfolger-Knoten (der Elter-Knoten ist direkter Vorgänger-Knoten, ein Kind-Knoten ist direkter Nachfolge-Knoten seines Elters – bei Beziehungen über mehrere Ebenen hinweg lässt man das „direkt“ weg). Ein Elter kann prinzipiell beliebig viele Kinder haben, ein Kind hat aber immer genau einen Elter. Einen Knoten ohne Kind nennt man Blatt. Einen Knoten mit allen seinen Nachfolgern nennt man Unterbaum. Den Knoten ohne Elter nennt man Wurzel. In der in der Informatik üblichen Darstellung ist der Wurzel-Knoten ganz oben. Mehrere Bäume (deren Knoten untereinander nicht verbunden sind) nennt man einen Wald. Mit diesen Begriffen kann man nachträglich einen Baum als Wurzel mit einem (evtl. leeren) Wald seiner Unterbäume beschreiben. Sind diese Unterbäume angeordnet (wie z. B. beim Inhaltsverzeichnis eines Buches), spricht man von einem geordneten Baum, sind sie dies nicht (wie z. B. in der Hierarchie einer Firma), spricht man von einem ungeordneten Baum.

*Wald, ((un-) geordn.) Baum, Unterbaum, Knoten, Kanten, Elter, Kind, Geschwister, (direkter) Vorgänger / Nachfolger, Wurzel, Blatt*

[9] Seit einiger Zeit hat auch hier die Gleichberechtigung Einzug gehalten: War früher ausschließlich von Vater, Söhnen und Brüdern die Rede, geht man heute mehr und mehr auf die neutrale Bezeichnung über – nichtsdestotrotz werden Sie in vielen Büchern noch die alten Bezeichnungen Vater und Sohn finden.

Der in der Informatik am häufigsten verwendete Baum ist der Binärbaum: dies ist ein geordneter Baum, bei dem jeder Knoten genau zwei (evtl. leere) Unterbäume hat. Man unterscheidet hier also jeweils den linken und rechten Unterbaum. *Binärbaum*

Die vielleicht wichtigste Eigenschaft eines Baumes ist, dass es von der Wurzel zu jedem Knoten im Baum genau einen Weg gibt.

Was uns noch fehlt, ist die Datenstruktur – wir beschränken uns hier auf Binärbäume, deren Knoten mit einem DataType beschriftet sind:[10]

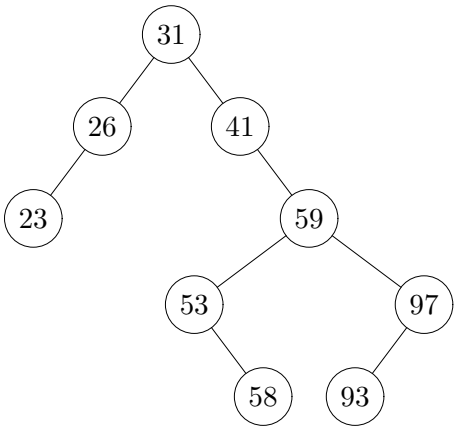
```
struct bTree;
typedef bTree * btPtr; // pointer to a binary tree
struct bTree {
  DataType value;
  btPtr left, right;
};
```

Für allgemeine Bäume (die für jeden Knoten eine variable Anzahl von Unterbäumen zulassen) müsste man zu jedem Knoten eine Liste der Wurzelknoten seiner Unterbäume speichern. Jeder Knoten hat dann einen Zeiger auf den Wurzelknoten seines linkesten Unterbaums und einen Zeiger auf seinen rechten Geschwisterknoten (Details selbst überlegen). Implizit hat so jeder Knoten auch zwei Zeiger wie bei Binärbäumen – man spricht hier deshalb auch von der Binarisierung von Bäumen. Wir werden uns in dieser Vorlesung aber auf Binärbäume beschränken.

### 1.3.2 Binäre Suchbäume

Ein binärer Suchbaum ist ein knotenbeschrifteter Binärbaum mit der Eigenschaft, dass für jeden Knoten $v$ gilt, dass die Knotenbeschriftung jedes Knotens im linken Unterbaum von $v$ einen Wert (echt) kleiner dem von $v$ hat und die Knotenbeschriftung jedes Knotens im rechten Unterbaum von $v$ einen Wert (echt) größer dem von $v$ hat (zwei verschiedenen Knoten müssen also stets verschiedene Knotenbeschriftungen haben). Bei der Verwaltung von Daten benötigt man insbesondere Funktionen zum Suchen, Einfügen und Löschen. *binärer Suchbaum*

Betrachten wir folgenden binären Suchbaum mit den Elementen 31, 41, 59, 26, 53, 58, 97, 93, 23 (vergewissern Sie sich, dass die Suchbaumeigenschaft erfüllt ist):




[10] Wir arbeiten hier mit den Pointern wie schon bei Listen, wer dies nicht möchte, erhält analog mit **typedef struct** bTree { DataType value; **struct** bTree ∗left, ∗right; } bTree; eine gleichermaßen in C wie in C++ funktionierende Variante.

Die Suchbaumeigenschaft führt direkt auf den Suchalgorithmus: als Beispiel sei zu überprüfen, ob die 53 im Suchbaum vorhanden ist. Wir beginnen in der Wurzel, die 31 ist nicht die gesuchte 53, 53 ist größer, damit muss die 53 – wenn sie im Suchbaum vorhanden ist – im rechten Unterbaum sein; wir überprüfen dessen Wurzel (die 41), auch hier ist die 53 größer; wir überprüfen zunächst noch die 59 und finden in dessem linken Unterbaum schließlich die 53. Sucht man die 42, so würde man die 42 nacheinander mit der 31, 41, 59 und 53 vergleichen; wenn die 42 im Suchbaum vorhanden wäre, hätte sie dann im linken Unterbaum der 53 sein müssen, dieser ist aber leer, damit kann die 42 nicht im Suchbaum sein. Allgemein wird also von der Wurzel aus jeweils im linken oder rechten Unterbaum weitergesucht, bis entweder der Wert gefunden wurde oder der betreffende Unterbaum leer ist.

Das Einfügen ist sehr ähnlich – wir müssen danach garantieren, dass die Suchbaumeigenschaft erfüllt ist: als Beispiel soll die 84 eingefügt werden. Wie beim Suchen verfolgen wir die linken und rechten Unterbäume – hier entlang der Knoten 31, 41, 59, 97 und 93 und stellen dann fest, dass dessen linker Unterbaum leer ist: hier ist die 84 einzufügen. Allgemein wird also von der Wurzel aus jeweils im linken oder rechten Unterbaum weitergesucht, bis entweder der Wert gefunden wurde (dann wird er nicht nochmal eingefügt) oder der betreffende Unterbaum leer ist und der Wert dort als neue Wurzel des Unterbaums eingetragen werden kann.

Beim Löschen sind zwei Fälle zu unterscheiden. Fall 1: der zu löschende Wert steht in einem Blatt – dann kann der Wert ohne Verletzung der Suchbaumeigenschaft einfach entfernt werden (im obigen Beispiel z. B. die Knoten 23, 58 und 93). Fall 2: der zu löschende Wert steht nicht in einem Blatt – dann entsteht eine Lücke, die so mit einem anderen Wert des Suchbaums ausgefüllt werden muss, dass die Suchbaumeigenschaft erhalten bleibt. Fall 2a: hat der Knoten nur ein Kind (z. B. der Knoten 41), dann kann der entsprechende Unterbaum an dessen Stelle gehängt werden (die 59 wird Wurzel des rechten Unterbaums der 31). Fall 2b: hat der Knoten jedoch zwei Kinder (z. B. der Knoten 59), so muss ein passender Wert gefunden werden, so dass die Suchbaumeigenschaft erhalten bleibt: dieses kann entweder der nächstkleinere oder der nächstgrößere Wert sein[11] – die dort entstehende Lücke wird wieder nach obigen Fällen unterschieden. Soll also die 59 gelöscht werden, so kann man diese z. B. durch den nächstgrößeren Wert ersetzen (der linkeste Knoten im rechten Unterbaum, hier die 93); der verschobene Knoten war ein Blatt, kann also direkt gelöscht werden (da der verschobene Knoten als linkester Knoten des Unterbaums kein linkes Kind haben kann, kann alternativ hier nur Fall 2a auftreten, also dessen rechter Unterbaum eine Ebene nach oben gehängt werden).

Übungsaufgaben: Fügen Sie weitere Knoten in obigen Suchbaum ein, suchen Sie nach verschiedenen Knoten, die im Suchbaum vorkommen, und nach solchen, die nicht vorkommen. Erweitern Sie obiges Beispiel, so dass es Knoten gibt, deren Löschung nach dem Ersetzungsschritt noch ein Umhängen eines Unterbaums erfordert. Welchen Aufwand haben die Funktionen zum Suchen, Einfügen und Löschen in $O$-Notation? Übung

Wie bei der binären Suche lassen sich die drei Funktionen sehr elegant rekursiv formulieren, wir geben hier den entsprechenden C++-Code an.[12]

[11] Diese werden auch Inorder-Vorgänger bzw. Inorder-Nachfolger genannt – diese Begriffe werden im nächsten Abschnitt noch genau eingeführt.

[12] Die Verwendung der Funktion bsTree() (Zeile 11 im **struct**) – ein sogenannter Konstruktor – geht bereits einen Schritt in Richtung objektorientierter Programmierung: er übernimmt die Initialisierung der Variablen des **struct**. Die auf den ersten Blick etwas merkwürdig anmutende Syntax : value(v),left(l),right(r) stellt eine sogenannte Initialisierungsliste dar – sie ist (für den Moment zumindest) gleichbedeutend mit den Zuweisungen value=v; left=l; right=r; – für die Zeiger left und right wurde hier als Default-Wert NULL (also der leere Zeiger) angegeben. Man kann so bei der Generierung dynamischer Variablen die Initialisierung sehr elegant durchführen. All diese Konzepte sind Teil der Sprache C++ – in C gibt es keine solchen Konstruktoren, man würde dort nach der Anforderung des Speicherbereichs an der Stelle im Programm die Variablen des **struct** mit den entsprechenden Werten „von Hand“ initialisieren müssen.

suchbaum.h

```
#ifndef SUCHBAUM_H
#define SUCHBAUM_H

typedef int DataType;

struct bsTree;
typedef bsTree * bstPtr; // pointer to a
struct bsTree {          // binary search tree
  DataType value;
  bstPtr left,right;
  bsTree(DataType v,bstPtr l=NULL,bstPtr r=NULL): value(v),left(l),right(r) {}
}; // for easy initialization of dynamic variables generated with "new"

void bstInsert(bstPtr &, const DataType);
bool bstFind(const bstPtr, const DataType);
bool bstDelete(bstPtr &, const DataType);
#endif
```

suchbaum.cpp

```
#include <cstdlib>
#include "suchbaum.h"
#include <iostream>
using namespace std;

void bstInsert(bstPtr & root, const DataType value) {
  if (root==NULL) {
    root = new bsTree(value); // children become NULL by default
  } else if (value<root->value) {
    bstInsert(root->left,value);
  } else if (value>root->value) {
    bstInsert(root->right,value);
  } else { // it is ==
    cout << "Element " << value << " is already in the tree!" << endl;
  }
}

bool bstFind(const bstPtr root, const DataType value) {
  if (root==NULL) {
    return false;
  } else if (value<root->value) {
    return bstFind(root->left,value);
  } else if (value>root->value) {
    return bstFind(root->right,value);
  } else { // it is ==
    return true;
  }
}
```

```
30 bool bstDelete(bstPtr & root, const DataType value) {
     // returns true if <value> was deleted, false if it isn't in the tree
     if (root==NULL) {
       return false; // no, the element was not in the tree
     } else if (value<root->value) {
35     return bstDelete(root->left,value);
     } else if (value>root->value) {
       return bstDelete(root->right,value);
     } else { // it is == - <value> is in the tree, I'll delete it!
       if (root->left==NULL && root->right==NULL) {
40       // it's a leaf => delete it
         delete root;
         root=NULL;
       } else if (root->left==NULL) {
         // right subtree must be non-empty, delete root and shift subtree up
45       bstPtr tbd=root;
         root=root->right;
         delete tbd;
       } else if (root->right==NULL) {
         // left subtree must be non-empty, delete root and shift subtree up
50       bstPtr tbd=root;
         root=root->left;
         delete tbd;
       } else {
         // both subtrees are non-empty => find inorder successor s,
55       // put s to the actual position, shift right subtree of s up
         bstPtr iosucc=root->right; // root is not NULL
         bstPtr iosparent=root;
         while (iosucc->left!=NULL) {
           iosparent=iosucc;
60         iosucc=iosucc->left;
         } // now iosucc is the inorder successor of root
         if (iosparent!=root) { // iosucc is not right child of root
           iosparent->left=iosucc->right; // may be NULL, doesn't matter
           iosucc->right=root->right;
65       }
         iosucc->left=root->left;
         bstPtr tbd=root;
         root=iosucc; // root parent's child ptr will be corrected by the &
         delete tbd;
70     }
       return true; // yes, we deleted an element
     }
   }
```

Machen Sie sich klar, wie durch das Konzept der Referenzparameter ein neu erzeugtes Element beim Einfügen korrekt mit dessem Elter verknüpft wird. Dasselbe Prinzip gilt – in etwas komplizierterer Form – auch für das Löschen. Prinzipiell lassen sich auch diese Funktionen für das Suchen, Einfügen und Löschen ohne Rekursion mit **while**-Schleifen implementieren; der entstehende Code ist aber weder effizienter noch leichter verständlich – probieren Sie es aus.

### 1.3.3 Die mittlere Suchdauer in binären Suchbäumen

Dieser Abschnitt reicht über den eigentlichen Stoff des zweiten Semesters hinaus und gibt einen Einblick, wie Abschätzungen zur Laufzeit im Mittel aussehen können und welche Techniken dabei angewendet werden können.

In der Regel entstehen binäre Suchbäume durch Einfügen der Elemente in zufälliger Reihenfolge. Der Aufwand zur Suche kann je nach entstehendem Baum zwischen logarithmisch und linear vielen Schritten liegen. Wir wollen hier nun untersuchen, was wir im Mittel erwarten können.

Wir betrachten dazu den Erwartungswert der Summe der Level[13] aller $n$ Knoten eines zufälligen binären Suchbaums: $\mathrm{SL}(n)$ – die erwartete mittlere Suchdauer (Anzahl der nötigen Vergleiche) in einem Suchbaum mit $n$ Elementen ist dann: $\mathrm{MS}(n) = 2 \cdot \frac{\mathrm{SL}(n)}{n} - 1$ für die erfolgreiche Suche (für jedes der im Mittel zu erwartenden Level sind 2 Vergleiche notwendig – bis auf das letzte, auf dem das Element gefunden wird).

Wenn die Elemente in zufälliger Reihenfolge eingefügt wurden, dann haben wir das $i$-kleinste Element als Wurzel gewählt und werden $i-1$ Elemente in den linken Unterbaum einfügen und $n-i$ Elemente in den rechten Unterbaum; diese $i-1$ bzw. $n-i$ Elemente liegen dann ein Level tiefer als die Wurzel. Das $i$ ist zufällig, gleichverteilt aus dem Intervall 1 bis $n$ gewählt, der Erwartungswert ist das Mittel aus diesen $n$ verschiedenen Fällen. Dies führt auf die Rekursionsgleichung

$$\mathrm{SL}(n) = n + \frac{1}{n} \sum_{i=1}^{n} (\mathrm{SL}(i-1) + \mathrm{SL}(n-i))$$

$+n$, da $n-1$ Knoten ein Level tiefer eingefügt werden und die Wurzel mit 1 auf dem aktuellen Level zu berücksichtigen ist. Die Summe der Level in den Unterbäumen wird mit dieser Rekursionsgleichung automatisch mit erledigt – wir müssen uns darum nicht explizit kümmern. Die erste wichtige Beobachtung (und erster Trick, der bei Betrachtungen im Mittel immer wieder vorkommt): der Parameter von $\mathrm{SL}(i-1)$ läuft von 0 bis $n-1$, der von $\mathrm{SL}(n-i)$ läuft von $n-1$ bis 0, d. h., jeder Parameter kommt genau zwei Mal vor:

$$\mathrm{SL}(n) = n + \frac{2}{n} \sum_{i=0}^{n-1} \mathrm{SL}(i) \qquad (①)$$

Die Summe der $\mathrm{SL}(i)$ ist unangenehm. Der zweite Trick: wir haben mit Ersetzen von $n$ durch $n-1$ auch eine Gleichung für $\mathrm{SL}(n-1)$:

$$\mathrm{SL}(n-1) = n - 1 + \frac{2}{n-1} \sum_{i=0}^{n-2} \mathrm{SL}(i) \qquad (②)$$

In dieser kommt die Summe fast identisch vor. Wir können nun durch geschicktes Bilden einer Differenz der Gleichungen ① und ② die Summe loswerden – wir betrachten dazu die Differenz $n \cdot ① - (n-1) \cdot ②$:

$$n \cdot \mathrm{SL}(n) - (n-1) \cdot \mathrm{SL}(n-1) = n \cdot n - (n-1)(n-1) + 2 \cdot \mathrm{SL}(n-1)$$

[13] Die Wurzel des Baums hat Level 1, die direkten Nachfolger Level 2, usw.

Die unangenehme Summe fällt fast vollständig heraus! Fassen wir etwas zusammen und bringen die $(n-1)\cdot \mathrm{SL}(n-1)$ auf die rechte Seite:

$$n \cdot \mathrm{SL}(n) = 2n - 1 + (n+1) \cdot \mathrm{SL}(n-1)$$

Die Terme mit $\mathrm{SL}(\cdot)$ sehen noch nicht ähnlich genug aus, wir dividieren beide Seiten durch $n(n+1)$:

$$\frac{1}{n+1} \cdot \mathrm{SL}(n) = \frac{2n-1}{n(n+1)} + \frac{1}{n} \cdot \mathrm{SL}(n-1)$$

Nun können wir den zweiten Trick erneut anwenden und ersetzen $\frac{1}{n} \cdot \mathrm{SL}(n-1)$ durch $\frac{2(n-1)-1}{(n-1)n} + \frac{1}{n-1} \cdot \mathrm{SL}(n-2)$ usw. – es sind $\mathrm{SL}(0) = 0$ und $\frac{1}{2} \cdot \mathrm{SL}(1) = \frac{1}{2} \cdot 1 = \frac{2\cdot 1-1}{1\cdot(1+1)}$, somit erhalten wir

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

Wir splitten die Summe auf:

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

Betrachten wir zunächst den zweiten Teil der Summe: $\frac{1}{i(i+1)}$ lässt sich auch schreiben als $\frac{1}{i} - \frac{1}{i+1}$. Somit gilt

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n} \frac{1}{i} - \frac{1}{i+1} = 1 - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \cdots + \frac{1}{n-1} - \frac{1}{n} + \frac{1}{n} - \frac{1}{n+1} = 1 - \frac{1}{n+1}$$

Solche Summen von Differenzen, bei denen lediglich vom ersten und letzten Term jeweils ein Teil übrigbleibt nennt man Teleskopsumme. Nun können wir die Terme noch weiter vereinfachen und zusammenfassen ($-2+2$ als vorbereitender Trick, Indexverschiebung $i+1 \to i$ und Abspalten des letzten Summengliedes sowie Einsetzen des Ergebnisses der Teleskopsumme):

$$\frac{1}{n+1} \cdot \mathrm{SL}(n) = -2 + 2 + \sum_{i=2}^{n} \frac{2}{i} + \frac{2}{n+1} - 1 + \frac{1}{n+1} = \sum_{i=1}^{n} \frac{2}{i} - 3 + \frac{3}{n+1}$$

und mit Einsetzen der Harmonischen Reihe $H(n) = \sum_{i=1}^{n} \frac{1}{i}$ schließlich

$$\mathrm{SL}(n) = 2(n+1)H(n) - 3n$$

Die Harmonische Reihe lässt sich recht gut abschätzen, es gilt die Näherung $H(n) \approx \ln n + \gamma$ ($\gamma \approx 0.577$ ist die Eulersche Konstante). Wie anfangs erwähnt ist der Erwartungswert für die Anzahl der Vergleiche zur Suche in einem zufälligen binären Suchbaum $\mathrm{MS}(n) = 2 \cdot \frac{\mathrm{SL}(n)}{n} - 1$. Mit dem Logarithmus zur Basis 2 ausgedrückt erhalten wir die Aussage:

$$\mathrm{MS}(n) \approx 2.773 \log n - 4.691$$

Gegenüber dem bestmöglichen Fall (einem vollständig ausgeglichenen Suchbaum) ist dies lediglich etwa 39% schlechter.[14]

Übung

Das Verhalten binärer Suchbäume im Mittel lässt sich auch gut experimentell untersuchen und somit obige Abschätzungen verifizieren. Schreiben Sie dazu kleine Funktionen, die

[14] In der Literatur wird häufig nur die im Mittel zu betrachtende Anzahl der Level als $\mathrm{MS}(n)$ angegeben, so findet sich meist die Abschätzung $\mathrm{MS}(n) \approx 1.386 \log n - 1.846$.

- die Anzahl der Knoten in einem Suchbaum bestimmen (1 für die Wurzel + Anzahl im linken + Anzahl im rechten Unterbaum);
- die Summe der Level der Knoten in einem Suchbaum bestimmt (Ansatz ähnlich, es empfiehlt sich, einen zusätzlichen Parameter in der Rekursion zu verwenden, der jeweils das Level des gerade betrachteten Knoten angibt);
- die Höhe des Baumes bestimmt – also das größte vorkommende Level im Baum (man bestimme dazu die Höhe der beiden Unterbäume; der größere der beiden Werte + 1 für die Wurzel ist der gesuchte Wert).

Wenn Sie die Korrektheit Ihrer Funktionen an kleinen Beispielen von Hand überprüft haben, können Sie versuchen, obige Näherungsformeln zu verifizieren. Fügen Sie dazu Zufallszahlen in einen anfangs leeren binären Suchbaum und vergleichen die experimentell ermittelten Werte mit denen aus der Theorie. Führen Sie für ein festes $n$ jeweils 100 Versuche durch und bilden Sie Mittelwerte. Wie groß sind die Abweichungen zum theoretischen Mittel?

Übung

Wir wollen nun noch den Einfluss der Löschfunktion untersuchen. Schreiben Sie sich dafür ein Testprogramm, das zufällig Zahlen in einen Suchbaum einfügt und wieder löscht (Sie können hierzu z. B. ein $n$ fest vorgeben und jeweils eine Zufallszahl zwischen 1 und $n$ bestimmen – ist diese Zahl im Suchbaum vorhanden, so lösche sie, andernfalls füge sie ein). Bestimmen Sie nach $i \cdot n^2/20$, $1 \leq i \leq 20$ Operationen[15] die Werte für die Summe der Level und vergleichen Sie die Werte mit denen aus den ersten Experimenten (beachten Sie, dass nun die Anzahl der Knoten im Suchbaum schwankt).

Es ist zu beobachten, dass die Werte nach und nach steigen! Die Ursache ist in der Löschfunktion zu suchen: da immer der Inordernachfolger gewählt wird, werden die Suchbäume immer linkslastiger (man kann sogar zeigen, dass nach etwa $n^2$ solcher Operationen die Höhe der Bäume in der Ordnung $\sqrt{n}$ liegt – sich also deutlich von dem eigentlich erwarteten logarithmischen Verhalten unterscheiden). Wir versuchen nun noch, dieses Fehlverhalten zu vermeiden, indem beim Löschen eines Knotens mit 2 Kindern dieser jeweils zufällig durch den Inordervorgänger oder Inordernachfolger ersetzt wird. Schreiben Sie eine modifizierte Löschfunktion, die dieses leistet. Untersuchen Sie nun, wie sich die entsprechenden Werte in der Praxis verhalten.

Übung

### 1.3.4 Baumdurchläufe – Tiefensuche in Binärbäumen

Beim Backtracking haben wir nacheinander alle Knoten des Entscheidungsbaums besucht. Dabei haben wir die Wurzel und die (im Beispiel der gerechten Erbschaft zwei) Unterbäume nacheinander bearbeitet. Speziell bei Binärbäumen betrachtet man verschiedene Reihenfolgen, wie man die Knoten in einem Binärbaum durchlaufen kann. Das prinzipielle Vorgehen ist aber identisch dem, das wir beim Backtracking schon gesehen haben: man betrachtet rekursiv die Unterbäume. Die drei am häufigsten betrachteten Reihenfolgen unterscheiden sich lediglich darin, ob man den Wurzelknoten vor den Unterbäumen betrachtet (Preorder) oder nach den Unterbäumen (Postorder) oder nach dem linken und vor dem rechten Unterbaum (Inorder).

*Preorder, Inorder, Postorder*

[15] Sie können hier auch andere Intervalle wählen, Sie sollten nur insgesamt wenigstens $n^2$ Einfüge- und Lösch-Operationen durchführen, um die auftretenden Phänomene beobachten zu können. Beachten Sie, dass das Einfügen und Löschen im Mittel pro Operation nur $O(\log n)$ Schritte benötigt, das Bestimmen der Knotenanzahl, Höhe und Summe der Level jedoch $O(n)$. Um die Gesamtlaufzeit des Tests nicht zu stark anwachsen zu lassen, sollte die Auswertung höchstens alle $n$ Operationen durchgeführt werden.

suchbaum.h (Fortsetzung)

```
void preorder(const bstPtr);
void inorder(const bstPtr);
void postorder(const bstPtr);
```

suchbaum.cpp (Fortsetzung)

```
void preorder(const bstPtr root) {
  if (root!=NULL) {
    // hier Knoten "bearbeiten"
    preorder(root->left);
    preorder(root->right);
  }
}

void inorder(const bstPtr root) {
  if (root!=NULL) {
    inorder(root->left);
    // hier Knoten "bearbeiten"
    inorder(root->right);
  }
}

void postorder(const bstPtr root) {
  if (root!=NULL) {
    postorder(root->left);
    postorder(root->right);
    // hier Knoten "bearbeiten"
  }
}
```

Der Aufwand ist bei allen drei Varianten linear, also $O(n)$. Die rekursive Funktion wird für jeden Knoten des Binärbaums genau ein Mal aufgerufen.

Wichtige Eigenschaften und Anwendungsfälle:

- Preorder: fügt man die Knoten eines Suchbaumes in dessen Preorder-Reihenfolge in einen anfangs leeren Suchbaum wieder ein, so erhält man exakt denselben Suchbaum – d. h., zum Abspeichern eines Suchbaums reicht es, dessen Preorder zu sichern.
- Inorder: bei Suchbäumen ist die Inorder-Reihenfolge stets aufsteigend sortiert[16] (überlegen Sie selbst, warum dies so ist). Damit haben wir sofort auch einen sehr einfachen Sortieralgorithmus: füge die zu sortierenden Elemente nacheinander in einen anfangs leeren Suchbaum ein und gib diesen dann in Inorder-Reihenfolge aus (der Aufwand beträgt – wie bei anderen einfachen Sortieralgorithmen auch – $O(n^2)$[17]).
- Postorder: viele Eigenschaften von Binärbäumen (z. B. die Anzahl seiner Blätter oder die maximale Anzahl der Knoten auf einem Weg von der Wurzel zu einem Blatt (die Höhe des Baums)) lassen sich mit einen Postorder-Durchlauf berechnen. Darüberhinaus werden

[16] Dies ist auch der Hintergrund, warum man beim nächstkleineren und nächstgrößerem Element im Suchbaum vom Inorder-Vorgänger bzw. Inorder-Nachfolger spricht.

[17] Etwas genauer: die Laufzeit hängt von der mittleren Tiefe der entstehenden binären Suchbäume ab – diese beträgt $O(\log n)$ wie wir im vorherigen Abschnitt gesehen haben – der Aufwand dieses einfachen Sortieralgorithmus beträgt im Mittel somit nur $O(n \log n)$.

z. B. vom Compiler arithmetischer Ausdrücke mittels Postorder ausgewertet (Stichwort Rechenbäume).

### 1.3.5 Visualisierung von Binärbäumen in der Konsole – Breitensuche in Binärbäumen

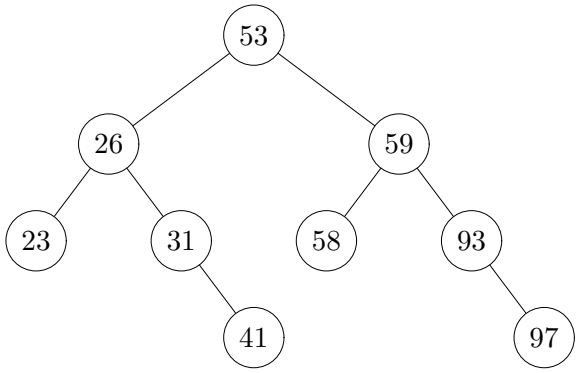
Eine weitere naheliegende Reihenfolge der Aufzählung der Knoten eines Binärbaums ist die Levelorder. Hierbei werden die Knoten Ebene für Ebene (beginnend mit der Wurzel) ausgegeben. Wir gehen hier nicht näher darauf ein, erwähnen lediglich den Zusammenhang von Levelorder zu der Datenstruktur Queue (die im vorherigen Abschnitt vorgestellten Reihenfolgen Preorder, Inorder und Postorder lassen sich alternativ zur Rekursion mit Hilfe eines Stacks realisieren). *Levelorder*

Die in der Vorlesung zur Verfügung gestellte Visualisierung von Binärbäumen basiert auf der Ausgabe der Knoten in Levelorder. Der C++-Code wird hier nicht weiter besprochen, ist der Vollständigkeit halber aber im Anhang A abgedruckt.

### 1.3.6 Optimale Suchbäume – ein typisches Beispiel für die Entwurfsstrategie Dynamisches Programmieren

Binäre Suchbäume werden zu einer Liste entarten, wenn in der Wurzel jedes Unterbaums der kleinste oder größte Wert des Unterbaums steht – ein einfaches solches Beispiel entsteht, wenn die Zahlen in aufsteigender (oder absteigender) Reihenfolge in den binären Suchbaum eingefügt werden (überlegen Sie sich selbst weitere Beispiele).

Statt die Positionen der Elemente im Baum dem Zufall der Einfügereihenfolge zu überlassen, kann man versuchen, die Elemente so anzuordnen, dass die Suchzeiten möglichst gering werden. Ein erster einfacher Ansatz besteht darin, die binäre Suche so auf Bäume zu übertragen, dass die Wurzel des Unterbaums jeweils dem mittleren Element der entsprechende Teilmenge entspricht. Es ist leicht einzusehen, dass auf diese Weise die maximale Tiefe im Suchbaum minimal wird (man spricht hierbei auch von einem ausgeglichenen (Such-)Baum). Hier das Beispiel aus Abschnitt 1.3.2 als ausgeglichener Suchbaum (die Aufteilung in linke und rechte Unterbäume entspricht der binären Suche, wenn die Elemente in einem Array wären): *ausgeglichener Suchbaum*



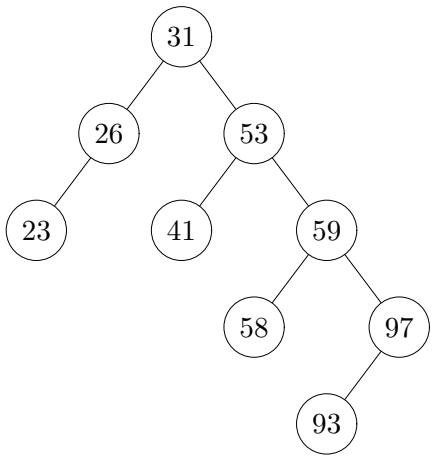

**Ist dies optimal?** Wenn sich die Zugriffshäufigkeiten auf einzelne Elemente stark voneinander unterscheiden, kann es sinnvoll sein, die Elemente des Suchbaums anders anzuordnen.

Wir betrachten dazu die (gewichtete) mittlere Suchdauer – diese gibt an, wieviel Ebenen des Suchbaums zur Suche eines Elements im Mittel betrachtet werden müssen. Jedes Element wird dazu mit seiner relativen Zugriffshäufigkeit (oder Zugriffswahrscheinlichkeit) gewichtet (gewichtetes arithmetisches Mittel). Sei $l(v)$ das Level, auf dem sich der Knoten $v$ befindet (die Wurzel hat Level 1), und $h(v)$ die relative Zugriffshäufigkeit (d. h., $\sum_v h(v) = 1$), so ist die mittlere Suchdauer $s = \sum_v h(v) \cdot l(v)$.

*mittlere Suchdauer*

Wird auf alle Elemente gleich häufig zugegriffen (relative Zugriffshäufigkeit jeweils $h(\cdot) = \frac{1}{9}$), so beträgt die mittlere Suchdauer $\frac{1}{9} \cdot (1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 2 \cdot 4) = 2\frac{7}{9}$ (ein Knoten befindet sich auf Ebene 1, zwei auf Ebene 2, vier auf Ebene 3 und zwei auf Ebene 4).

Nehmen wir an, die Zugriffshäufigkeiten seien $h(23) = 0.10$, $h(26) = 0.16$, $h(31) = 0.25$, $h(41) = 0.20$, $h(53) = 0.10$, $h(58) = 0.03$, $h(59) = 0.02$, $h(93) = 0.08$, $h(97) = 0.06$, so ergibt sich im obigen ausgeglichenen Suchbaum eine mittlere Suchdauer von $s = 0.10 \cdot 1 + (0.16 + 0.02) \cdot 2 + (0.10 + 0.25 + 0.03 + 0.08) \cdot 3 + (0.20 + 0.06) \cdot 4 = 2.88$. Folgender (nicht ausgeglichener) Suchbaum hat jedoch eine mittlere Suchdauer von $s = 0.25 \cdot 1 + (0.16 + 0.10) \cdot 2 + (0.10 + 0.20 + 0.02) \cdot 3 + (0.03 + 0.06) \cdot 4 + 0.08 \cdot 5 = 2.49$, also deutlich weniger als im naheliegenden ausgeglichenen Suchbaum. Geht es noch besser?




Ein Suchbaum ist optimal, wenn kein anderer Suchbaum (mit denselben Elementen und Zugriffshäufigkeiten) eine geringere mittlere Suchdauer hat.

*optimaler Suchbaum*

**Optitimalitätseigenschaft optimaler Suchbäume:** Jeder Unterbaum eines optimalen Suchbaums ist (für die zugehörige Teilmenge der Elemente) ein optimaler Suchbaum – wenn dies nicht so wäre, könnte man den entsprechenden Unterbaum durch einen mit geringerer mittlerer Suchdauer ersetzen und erhielte so auch für den gesamten Suchbaum eine geringere mittlere Suchdauer.

Diese Eigenschaft führt zu folgendem Algorithmus: Wir berechnen die optimalen Suchbäume für immer längere Teilfolgen der Elemente. Die optimalen Suchbäume für die einelementigen Teilfolgen sind trivial (der optimale Suchbaum besteht nur aus dem einen Element). Bei den zweielementigen Teilfolgen kann entweder das kleinere oder das größere Element die Wurzel sein (der andere Knoten wird zum rechten bzw. linken Unterbaum) – wir berechnen beide Werte für die jeweilige mittlere Suchdauer und wählen den kleineren aus. Bei den dreielementigen Teilfolgen kommen drei Elemente als Wurzel in Frage (je nach Wahl sind die Unterbäume leer, bestehen aus 1 oder aus 2 Knoten) – wir berechnen die drei Werte für die jeweilige mittlere

Suchdauer und wählen den kleinsten aus. Analog wird für entsprechend längere Teilfolgen vorgegangen.

Formal lässt sich dies sehr kompakt formulieren: Seien $a_1, a_2, \ldots, a_n$ die (der Größe nach sortierten) Elemente im Suchbaum. Sei weiter $s(i,j)$ die kleinste mittlere Suchdauer für den Suchbaum, der aus den Elementen $a_i, a_{i+1}, \ldots, a_j$ besteht (ist $i > j$, so ist der Suchbaum leer und $s(i,j) = 0$), und $w(i,j)$ dessen Wurzel. Sei ferner $g(i,j) = \sum_{k=i}^{j} h(a_k)$ die Summe der relativen Häufigkeiten der Elemente $a_i$ bis $a_j$. Dann sind

$$s(i,j) = \min\{s(i,k-1) + s(k+1,j) + g(i,j) \mid k \in \{i,\ldots,j\}\}$$

und $w(i,j)$ das $k$, für das das Minimum angenommen wurde (bei Gleichheit für mehrere Werte wählen wir einen nach Belieben aus). Das „$+g(i,j)$“ kommt daher, dass alle Elemente des linken und rechten Unterbaums jeweils ein Level tiefer stehen und die Wurzel auf Level 1.

Überlegen Sie sich, warum es reicht, jeweils die zusammenhängenden Teilfolgen zu betrachten. Übung

Wir betrachten den Ablauf an obigem Beispiel: die relativen Zugriffshäufigkeiten der Elemente des Suchbaums:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $a_i$ | 23 | 26 | 31 | 41 | 53 | 58 | 59 | 93 | 97 |
| $h(a_i)$ | 0.10 | 0.16 | 0.25 | 0.20 | 0.10 | 0.03 | 0.02 | 0.08 | 0.06 |

Die Gewichte $g(i,j)$ (wir sehen, dass hier nur die eine Hälfte der Matrix benötigt wird):

| i\j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.10 | 0.26 | 0.51 | 0.71 | 0.81 | 0.84 | 0.86 | 0.94 | 1.00 |
| 2 | | 0.16 | 0.41 | 0.61 | 0.71 | 0.74 | 0.76 | 0.84 | 0.90 |
| 3 | | | 0.25 | 0.45 | 0.55 | 0.58 | 0.60 | 0.68 | 0.74 |
| 4 | | | | 0.20 | 0.30 | 0.33 | 0.35 | 0.43 | 0.49 |
| 5 | | | | | 0.10 | 0.13 | 0.15 | 0.23 | 0.29 |
| 6 | | | | | | 0.03 | 0.05 | 0.13 | 0.19 |
| 7 | | | | | | | 0.02 | 0.10 | 0.16 |
| 8 | | | | | | | | 0.08 | 0.14 |
| 9 | | | | | | | | | 0.06 |

Die einelementigen optimalen Suchbäume mit ihren Wurzeln und mittleren Suchdauern:

| $i,j$ | 1,1 | 2,2 | 3,3 | 4,4 | 5,5 | 6,6 | 7,7 | 8,8 | 9,9 |
|---|---|---|---|---|---|---|---|---|---|
| $s(i,j)$ | 0.10 | 0.16 | 0.25 | 0.20 | 0.10 | 0.03 | 0.02 | 0.08 | 0.06 |
| $w(i,j)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Die zweielementigen optimalen Suchbäume mit ihren Wurzeln und mittleren Suchdauern (das Element mit der größeren Zugriffshäufigkeit ist jeweils die Wurzel – das Programm wird jedoch stur beide mittleren Suchdauern ausrechnen, am Beispiel $(i,j) = (1,2)$: mit Element 1 als Wurzel ist der linke Unterbaum leer (nach obiger Formel $s(1,0) = 0$), der rechte besteht aus Element 2 ($s(2,2) = 0.16$), somit in der Summe $0.00 + 0.16 + g(1,2) = 0.42$; mit Element 2 als Wurzel betrachtet man $s(1,1) + s(3,2) + g(1,2) = 0.10 + 0.00 + 0.26 = 0.36$; somit ergeben sich die Werte $s(1,2) = 0.36$ und $w(1,2) = 2$):

| $i,j$ | 1,2 | 2,3 | 3,4 | 4,5 | 5,6 | 6,7 | 7,8 | 8,9 |
|---|---|---|---|---|---|---|---|---|
| $s(i,j)$ | 0.36 | 0.57 | 0.65 | 0.40 | 0.16 | 0.07 | 0.12 | 0.20 |
| $w(i,j)$ | 2 | 3 | 3 | 4 | 5 | 6 | 8 | 8 |

Die dreielementigen optimalen Suchbäume mit ihren Wurzeln und mittleren Suchdauern (auch hier als Beispiel die zu berechnenden Werte für $(i,j) = (1,3)$: mit Element 1 als Wurzel betrachtet man $s(1,0)+s(2,3)+g(1,3) = 0.00+0.57+0.51 = 1.08$, mit Element 2 als Wurzel $s(1,1)+s(3,3)+g(1,3) = 0.10+0.25+0.51 = 0.86$, mit Element 3 $s(1,2)+s(4,3)+g(1,3) = 0.36+0.00+0.51 = 0.87$; somit ergeben sich die Werte $s(1,3) = 0.86$ und $w(1,3) = 2$):

| $i,j$ | 1,3 | 2,4 | 3,5 | 4,6 | 5,7 | 6,8 | 7,9 |
|---|---|---|---|---|---|---|---|
| $s(i,j)$ | 0.86 | 0.97 | 0.90 | 0.49 | 0.22 | 0.20 | 0.24 |
| $w(i,j)$ | 2 | 3 | 4 | 4 | 5 | 8 | 8 |

Die vier- und fünfelementigen Teilmengen:

| $i,j$ | 1,4 | 2,5 | 3,6 | 4,7 | 5,8 | 6,9 | 1,5 | 2,6 | 3,7 | 4,8 | 5,9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s(i,j)$ | 1.27 | 1.27 | 0.99 | 0.57 | 0.43 | 0.32 | 1.57 | 1.39 | 1.07 | 0.83 | 0.57 |
| $w(i,j)$ | 3 | 3 | 4 | 4 | 5 | 8 | 3 | 3 | 4 | 5 | 8 |

Und die verbleibenden Teilmengen.

| $i,j$ | 1,6 | 2,7 | 3,8 | 4,9 | 1,7 | 2,8 | 3,9 | 1,8 | 2,9 | 1,9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $s(i,j)$ | 1.69 | 1.49 | 1.36 | 1.01 | 1.79 | 1.83 | 1.56 | 2.13 | 2.04 | 2.37 |
| $w(i,j)$ | 3 | 3 | 4 | 5 | 3 | 3 | 4 | 3 | 4 | 3 |

Der optimale Suchbaum hat also eine mittlere Suchdauer von 2.37 und hat die Wurzel $a_3 = 31$ – der oben angegebene Baum war also schon recht nah am Optimum.

In obiger Berechnung war die Wurzel der optimalen Teilbäume jeweils eindeutig. Ist dies nicht der Fall, so darf beliebig unter den Wurzeln, die zu den minimalen mittleren Suchdauern führen, gewählt werden.

Nun fehlt nur noch der C/C++-Code: Die Indizes der Arrays laufen – wie in C/C++ üblich – von 0 bis $n-1$ (statt wie im Beispiel von 1 bis $n$). Ein Problem sind die zweidimensionalen Arrays, die während der Laufzeit benötigt werden, deren Größen aber zur Compile-Zeit nicht bekannt sind und deshalb dynamisch erzeugt werden müssen. Die naheliegende Möglichkeit **double** s[size][size]; funktioniert zwar mit dem GNU-C/C++-Compiler, entspricht aber nicht dem Sprachstandard (und wird z. B. auch im Microsoft Visual Studio C++ nicht akzeptiert)! Wir generieren uns zunächst ein Array aus Zeigern und belegen diese dann jeweils mit Zeigern auf Arrays des benötigten Datentyps. Das Löschen funktioniert analog (zunächst werden die Arrays des Datentyps gelöscht und dann das Array der Zeiger). Das Prinzip ist für beliebige Datentypen stets gleich und so kann man sich dazu zwei Makros definieren, die (quadratische) zweidimensionale Arrays erzeugen und wieder aus dem Speicher entfernen:

```
#define CreateDyn2DArray(typ,name,size) typ **name=new typ*[size]; \
                    for(int i=0;i<size;i++) name[i]=new typ[size];
#define DeleteDyn2DArray(name,size) for(int i=0;i<size;i++) delete name[i]; \
                                    delete name;
```

Die Werte $g(i,j)$ lassen sich nun leicht mit zwei geschachtelten Schleifen vorab berechnen, die $s(i,j)$ müssen für immer größere Differenzen $j-i$ berechnet werden. Bei der Umsetzung in C/C++ muss man noch beachten, dass bei den Fällen, dass der linke oder rechte Unterbaum leer ist, bei der Berechnung auf Werte $s(i,j)$ mit $i > j$ zugegriffen würde – man kann dies im Programm so umgehen, dass man für die Minimumsuche diese beiden Randfälle miteinander vergleicht und in der Schleife die restlichen Fälle betrachtet werden.

Um aus den Werten den Baum zu konstruieren geht man – wie bei Bäumen üblich – rekursiv vor: $k = w(1,n)$ ist die Wurzel des optimalen Suchbaums, damit gehören die Elemente $a_1,\ldots,a_{k-1}$ zum linken Unterbaum, dessen Wurzel ist $k' = w(1,k-1)$. Die Unterbäume von

$a_{k'}$ haben die Wurzeln $w(1, k'-1)$ und $w(k'+1, k-1)$ ... in welcher Reihenfolge wird hier der optimale Suchbaum aus den berechneten Daten rekonstruiert? Übung

`suchbaum.h` (Fortsetzung)

```
// berechne die Wurzeln der Unterbäume eines optimalen binären Suchbaums:
bstPtr optBST(int[], double[], const int); // Werte, Wahr'keiten, Anzahl
// generiere den Baum mit den Werten aus den berechneten Wurzeln
bstPtr genBST(int[], int**, const int, const int, const int);
// Aufruf genBST(Wertearray, Wurzelarray, Anzahl, linker, rechter Rand);
```

`suchbaum.cpp` (Fortsetzung)

```
bstPtr optBST(int elems[], double probs[], const int size) {
  CreateDyn2DArray(int,w,size);     // w[x][y]=Wurzel für Indizes x bis y
  CreateDyn2DArray(double,s,size); // s[x][y]=mittlere Suchdauer x bix y
  CreateDyn2DArray(double,g,size); // g[x][y]=Summe Wahr'keiten  x bis y
  int i,j,groesse,links,rechts,auswahl;
  double bmsd,vmsd; // bisher kleinste und zu vergleichende mittlere Suchdauer
  for(i=0;i<size;i++) {
    s[i][i]=probs[i];
    w[i][i]=i;
    g[i][i]=probs[i];
    for(j=i+1;j<size;j++) {
      g[i][j]=g[i][j-1]+probs[j];
    }
  }
  for(groesse=2;groesse<=size;groesse++) {
    for(links=0;links+groesse<=size;links++) {
      rechts=links+groesse-1;
      // Minimumsuche mit einem der Fälle "leerer Unterbaum" initialisieren
      bmsd=s[links+1][rechts]; vmsd=s[links][rechts-1];
      if(vmsd<bmsd) {
        bmsd=vmsd; w[links][rechts]=rechts;
      } else {
        w[links][rechts]=links;
      }
      for(auswahl=links+1;auswahl<=rechts-1;auswahl++) {
        vmsd=s[links][auswahl-1]+s[auswahl+1][rechts];
        if(vmsd<bmsd) {
          bmsd=vmsd; w[links][rechts]=auswahl;
        }
      }
      s[links][rechts]=bmsd+g[links][rechts]; // g[l][r] erst hier addiert
    }
  }
  bstPtr erg=genBST(elems,w,size,0,size-1);
  DeleteDyn2DArray(w,size);
  DeleteDyn2DArray(s,size);
  DeleteDyn2DArray(g,size);
  return erg;
}
```

```
bstPtr genBST(int elems[],int **roots,const int size,const int l,const int r){
  if (l>r) {
    return NULL;
  } else {
45  int rootidx=roots[l][r];
    bstPtr root=new bsTree(elems[rootidx]);
    root->left=genBST(elems,roots,size,l,rootidx-1);
    root->right=genBST(elems,roots,size,rootidx+1,r);
    return root;
50  }
}
```

Die systematische Berechnung der optimalen Lösung aus schon berechneten optimalen Teillösungen haben wir bereits unter dem Namen *Dynamisches Programmieren* kennengelernt. Beim Beispiel der Optimalen Suchbäume ist der typische Aufbau der drei geschachtelten Schleifen (äußere Schleife über die Größe der betrachteten Teilprobleme, mittlere Schleife über die verschiedenen Teilprobleme der in der äußeren Schleife betrachteten Größe des Teilproblems, innere Schleife die Auswahl aus den verschiedenen möglichen Lösungen) besonders deutlich. Ebenfalls typisch ist hier der kubische Aufwand – die Berechnung der optimalen Suchbäume benötigt mit obigem Verfahren

$$\sum_{g=1}^{n}\sum_{l=0}^{n-g} g = \sum_{g=1}^{n}(n-g)g = n\cdot\frac{n(n-1)}{2} - \frac{n(n+1)(2n+1)}{6} \in O(n^3)$$

Schritte (ein Schritt entspricht hier der Ausführung der Zeilen 26 bis 29).

Der Aufwand kann in der Praxis noch verringert werden: Für die Wurzel eines optimalen Suchbaums kommen nicht alle Elemente in Frage: die Wurzel $w(i,j)$ kann nur im Bereich $w(i,j-1)$ bis $w(i+1,j)$ liegen – diese Eigenschaft nennt man „Monotonie der Wurzeln“. Es reicht dazu Zeile 25 zu ändern (**for**(auswahl=w[links][rechts−1];auswahl<=w[links+1][rechts];auswahl++)), die Initialisierung vor der Schleife anzupassen und darauf zu achten, dass kein Zugriff auf w[i][j] mit i>j stattfindet. Der formale Beweis für die Korrektheit dieser verkürzten Berechnung ist aufwendig. Der Laufzeit reduziert sich trotz dieser nur kleinen Änderung auf

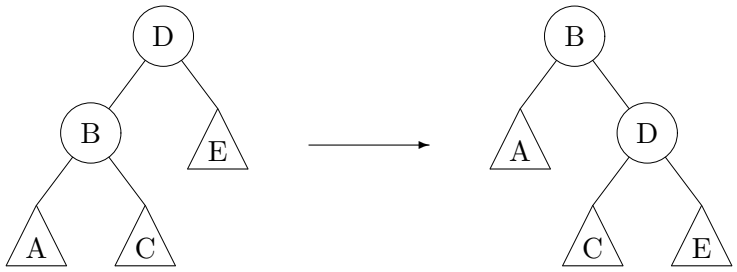
$$\begin{aligned}
& \sum_{g=1}^{n}\sum_{l=0}^{n-g}(1+w(l+1,l+g-1)-w(l,l+g-2)) \\
= & \sum_{g=1}^{n}(n-g+1+w(n-g+1,n-1)-w(0,g-2)) \\
\leq & \sum_{g=1}^{n}(n-g+1+n-1) \\
\leq & n^2-\frac{n(n-1)}{2}+n+n^2-n \\
\in & O(n^2)
\end{aligned}$$

**Schlussbemerkung:** das weitere Einfügen und/oder Löschen von Elementen zerstört in der Regel die Optimalitätseigenschaft – das Wiederherstellen dieser Eigenschaft ist zu aufwendig, als dass man dies häufig durchführen kann. Optimale Suchbäume finden daher nur bei statischen Daten Verwendung wie z. B. bei Lexika oder der Erkennung von Schlüsselwörtern durch einen Compiler.

### 1.3.7 Rotation in binären Suchbäumen

Speziell in sich dynamisch oft ändernden Datenbeständen müssen neue Methoden gefunden werden, um die Suchzeiten gering zu halten. Man könnte daher versuchen, die binären Suchbäume mit möglichst geringem Aufwand so umzuordnen, dass diese ausreichend ausgeglichen bleiben. Es gibt hierzu verschiedene Ansätze, die auf dem gemeinsamen Prinzip der Rotation beruhen.

*Rotation*

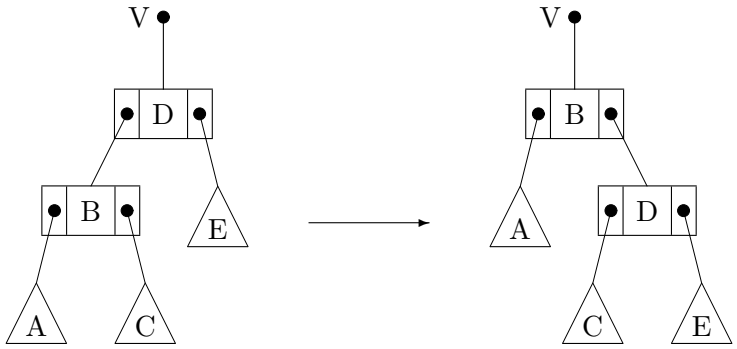



Diese Umordnung nennt man eine Rechtsrotation (entlang der Kante B—D). Die umgekehrte Richtung ist eine Linksrotation (ebenfalls entlang der Kante B—D). Man beobachte, dass dabei die Suchbaumeigenschaft erhalten bleibt.

*Links-, Rechtsrotation*

Zur Erinnerung: die Werte in den Unterbäumen sind angeordnet (im linken Unterbaum befinden sich die Werte, die kleiner als die Wurzel sind, im rechten Unterbaum die größeren).

**Implementierung in C++** Zunächst scheint dies nicht ganz trivial zu sein – folgende Darstellung, die etwas näher an der Implementierung ist, zeigt jedoch, dass dem nicht so ist:




Es werden bei der Rotation nur drei Variablen geändert (PtrV (der Zeiger auf die Wurzel des (Unter-)Baums), PtrD−>left und PtrB−>right), wobei nur drei Werte vorkommen (die Adressen der Knoten D, B und C; also die in den Variablen PtrD, PtrB und PtrC abgelegten Adressen). Vor der Rotation gilt PtrV==PtrD, PtrD−>left==PtrB und PtrB−>right==PtrC; nach der Rotation gilt PtrV==PtrB, PtrD−>left==PtrC und PtrB−>right==PtrD.

Man sieht, dass bei der Rechtsrotation lediglich die Adressen, die in den drei Zeigern PtrV, PtrD−>left und PtrB−>right gespeichert sind, „im Kreis“ vertauscht werden müssen. Zur Erinnerung: eine Zuweisung bedeutet bei Zeigern, dass die angegebene Speicheradresse kopiert wird und nicht der Inhalt, der an dieser Adresse steht.

```
void Rotate(bstPtr &p1, bstPtr &p2, bstPtr &p3) {
  bstPtr ph=p1; p1=p2; p2=p3; p3=ph;
}
```

Sei root die Zeiger-Variable auf die Wurzel des Unterbaums (dies entspricht der obigen Variablen PtrV), dann lautet der Aufruf bei der Rechtsrotation

```
Rotate(root,root->left,root->left->right);
```

– man beachte, wie hier durch das Referenzparameter-Konzept die Zeiger verändert werden. Bei einer Linksrotation verläuft es analog, der Aufruf lautet dann

```
Rotate(root,root->right,root->right->left);
```

– der Aufwand besteht offensichtlich nur aus einer konstanten Anzahl von Zeigerzuweisungen und ist damit unabhängig von der Größe der umgehängten Unterbäume konstant.

**Ausblick:** Unter `http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree` findet man eine Übersicht verschiedener Methoden, um binäre Suchbäume ausreichend ausgeglichen zu halten. Wir stellen im nun folgenden Abschnitt die häufig verwendeten AVL-Bäume vor.

### 1.3.8 AVL-Bäume

*höhenbalancierte binäre Suchbäume*

AVL-Bäume wurden 1962 von G.M. Adelson-Velsky und E.M. Landis eingeführt und zeigen, wie man mit relativ geringem Aufwand binäre Suchbäume höhenbalanciert halten kann – d. h.: die Differenz zwischen der Höhe des rechten und linken Unterbaums, die Höhenbalance, darf dabei nur die Werte $-1$, 0 und $+1$ annehmen (diese Werte werden als Zusatzinformation im Knoten gespeichert). Ist diese Bedingung erfüllt, so lässt sich zeigen, dass die Höhe solch eines binären Suchbaums durch $O(\log n)$ beschränkt ist. Wie wir gleich sehen werden, ist damit auch der Aufwand für die drei Operationen Einfügen, Suchen und Löschen jeweils durch $O(\log n)$ beschränkt.

Die **Suchfunktion** ist identisch der in normalen binären Suchbäumen – während jene jedoch zu Listen entarten können und somit die Laufzeit nur durch $O(n)$ beschränkt ist, wird bei AVL-Bäumen die Einhaltung der Höhenbalance die Höhe der Bäume und damit die Laufzeit auf $O(\log n)$ reduzieren.

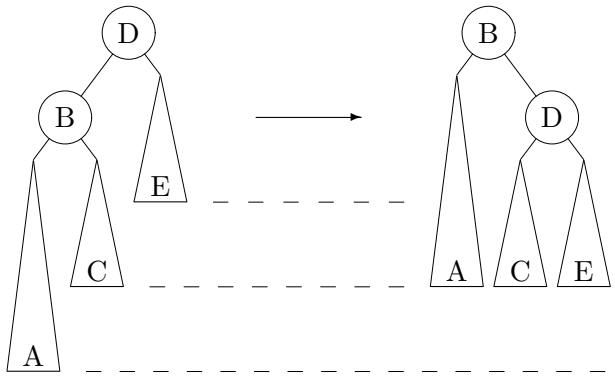
Beim **Einfügen** geht man bis zum Finden der leeren Stelle im Suchbaum ebenso vor wie gehabt. Nach dem Einfügen müssen nun noch die Höhenbalancen angepasst werden – dabei wird ausgehend vom soeben eingefügten Blatt jeweils für die direkten Vorgänger die Höhenbalance angepasst (es können sich grundsätzlich nur die Höhenbalancen der Knoten auf dem Weg von der Wurzel zum eingefügten Knoten verändern, machen Sie sich klar, warum dies so ist):

- Das eingefügte Blatt hat die Höhenbalance 0 (beide Unterbäume haben Höhe 0). Der Knoten hat die Höhe des zuvor leeren Unterbaums auf 1 erhöht, damit ändert sich die Höhenbalance des direkten Vorgängers:
  - Ist der Knoten Wurzel eines rechten Unterbaums, so hat sich die Höhenbalance des direkten Vorgängers um 1 erhöht.
  - Ist der Knoten Wurzel eines linken Unterbaums, so hat sich die Höhenbalance des direkten Vorgängers um 1 verringert.

- Hier können nun mehrere Fälle auftreten:
  - Die Höhenbalance wurde auf 0 geändert. Dann ist die Höhe dieses Unterbaumes unverändert geblieben (der Unter-Unterbaum, der zuvor eine Ebene weniger hatte, ist um eine Ebene erweitert worden). Das Einfügen ist damit beendet, es können sich keine weiteren Höhenbalancen geändert haben.
  - Die Höhenbalance ist auf $-1$ oder $+1$ geändert worden. Dann ist die Höhe dieses Unterbaumes um 1 erhöht worden (beide Unter-Unterbäume waren vorher gleich hoch). Wir müssen die Höhenbalance des direkten Vorgängers ebenfalls anpassen.
  - Die Höhenbalance ist auf $-2$ oder $+2$ geändert worden. Hier ist ein Reparaturschritt notwendig, der die Höhenbalancen im erlaubten Bereich belässt.

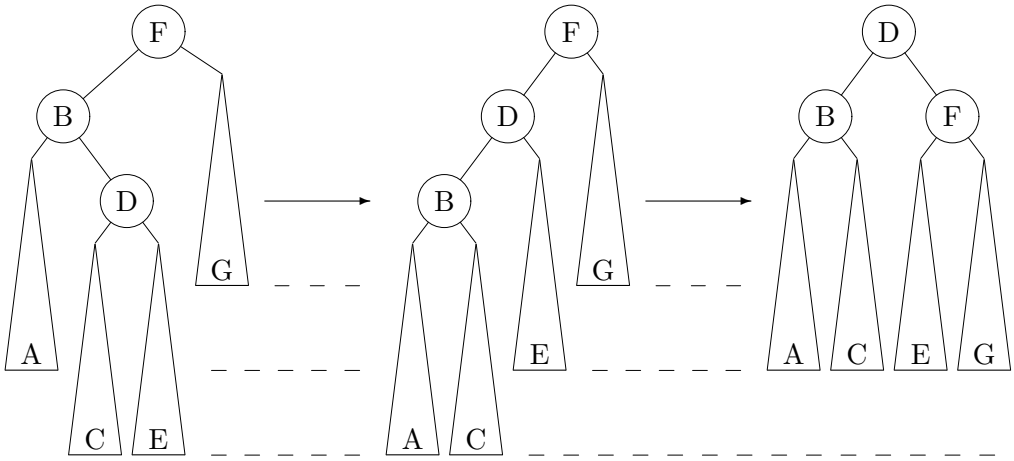
**Reparatur der Höhenbalancen:** Beim Einfügen können bei der Reparatur vier verschiedene Fälle auftreten. Der Unterbaum wird dabei durch eine Rotation oder eine Kombination aus zwei Rotationen so verändert, dass die Höhe des Unterbaums im Vergleich zum Zustand vor dem Einfügen unverändert geblieben ist. Wir werden also auch hier keine weiteren Höhenbalancen anpassen müssen.

- Fall 1: die Höhenbalance wurde auf $-2$ geändert und dessen linker Unterbaum hat eine Höhenbalance von $-1$, d. h., der Knoten wurde im linken Unterbaum des linken Unterbaums eingefügt (sonst wären die Höhenbalancen nicht so angepasst worden): Wir rotieren entlang der Kante B—D nach rechts (Rechtsrotation): *Rechtsrotation*




  Nach der Rotation hat der Unterbaum wieder die Höhe wie vorher (der Unterbaum A hatte vorher eine um 1 geringere Höhe). Die Höhenbalancen von B und D sind nun jeweils 0.

- Fall 2: die Höhenbalance wurde auf $-2$ geändert und dessen linker Unterbaum hat eine Höhenbalance von $+1$, d. h., der Knoten wurde im rechten Unterbaum des linken Unterbaums eingefügt: Wir rotieren zunächst entlang der Kante B—D nach links und anschließend entlang der Kante D—F nach rechts; man bezeichnet dieses als Doppelrotation (hier Linksrechtsrotation): *Doppelrotation, Linksrechtsrotation*




Genau genommen stimmt die Abbildung nicht ganz: es reicht nur genau einer der beiden Unterbäume C oder E bis zur letzten Ebene (warum?) – die Höhen der Unterbäume C und E ändern sich nicht, dementsprechend ist der eine Unterbaum C oder E auch nach der Doppelrotation eine Ebene weniger hoch. War die Höhenbalance von D=−1, so sind danach die Höhenbalancen von B 0 und von F +1; war die Höhenbalance von D=+1, so sind danach die Höhenbalancen von B −1 und von F 0. Die Höhenbalance von D ist danach in jedem Fall 0.

- Fall 3: ist symmetrisch zu Fall 1 – die Höhenbalance wurde auf +2 geändert und dessen rechter Unterbaum hat eine Höhenbalance von +1, d. h., der Knoten wurde im rechten Unterbaum des rechten Unterbaums eingefügt. Wir führen eine Linksrotation durch. *Linksrotation*

- Fall 4: ist symmetrisch zu Fall 2 – die Höhenbalance wurde auf +2 geändert und dessen rechter Unterbaum hat eine Höhenbalance von −1, d. h., der Knoten wurde im linken Unterbaum des rechten Unterbaums eingefügt: Wir führen eine Rechtslinksrotation durch: *Rechtslinksrotation*

**Wichtig:** nach einer Rotation oder Doppelrotation müssen keine weiteren Höhenbalancen mehr angepasst werden. Beim Einfügen eines beliebigen Elements ist maximal diese eine (Doppel-)Rotation notwendig – tritt keine Rotation auf, werden die Höhenbalancen ggf. bis zur Wurzel hin angepasst. Der Aufwand beträgt höchstens zwei mal die Höhe des AVL-Baums.

Nun fehlt nur noch das **Löschen**: Wie beim normalen Suchbaum werden auch hier nur Blätter gelöscht, d. h., ist der Knoten kein Blatt, so hat dieser entweder zwei Kinder (und wir ersetzen ihn durch seinen Inordernachfolger) oder er hat nur ein Kind (dieses muss dann – bei AVL-Bäumen – ein Blatt sein (warum?)), durch welches der zu löschende Knoten dann ersetzt wird. Auch hier wurde der Unterbaum durch das Löschen eines Blattes um eins verringert – wir müssen wieder auf dem Pfad der Vorgänger die Höhenbalancen anpassen. Analog um Einfügen können dabei folgende Fälle auftreten.

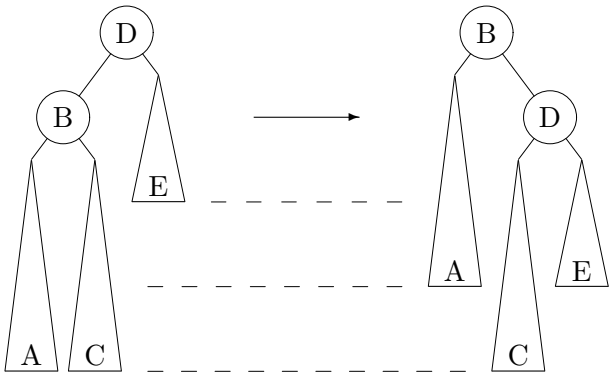
- Die Höhenbalance ist auf −1 oder +1 geändert worden (und war vorher 0). Dann ist die Höhe dieses Unterbaumes unverändert geblieben (nur einer der zuvor gleich hohen Unter-Unterbäume ist um eine Ebene verkleinert worden). Das Löschen ist damit beendet, es müssen keine weiteren Höhenbalancen angepasst werden.

- Die Höhenbalance wurde auf 0 geändert. Dann war zuvor die Höhe des Unterbaums gleich der des höheren der beiden Unter-Unterbäume (derjenige, in dem das Element gelöscht wurde). Wir müssen die Höhenbalance des direkten Vorgängers ebenfalls anpassen.

- Die Höhenbalance ist auf $-2$ oder $+2$ geändert worden. Hier ist ein Reparaturschritt notwendig, der die Höhenbalancen im erlaubten Bereich belässt.

**Reparatur der Höhenbalancen:** Ist die Höhenbalance auf $-2$ oder $+2$ geändert worden, so hat das Kind in dem Unterbaum, in dem das Element gelöscht wurde, stets die Balance 0 (warum?). Beim Löschen hängt die notwendige Rotation von der Balance des anderen Unter-Unterbaums ab.

- Fall 1: die Höhenbalance wurde auf $-2$ geändert und dessen linker Unterbaum hat eine Höhenbalance von $-1$ (der Knoten wurde aus dem rechten Unterbaum gelöscht). Es ist eine Rechtsrotation notwendig (Anpassung der Höhenbalancen wie beim Einfügen). Die Höhenbalance des direkten Vorgängers muss ebenfalls angepasst werden.
- Fall 2: die Höhenbalance wurde auf $-2$ geändert und dessen linker Unterbaum hat eine Höhenbalance von $+1$ (der Knoten wurde aus dem rechten Unterbaum gelöscht). Es ist eine Linksrechtsrotation notwendig (Anpassung der Höhenbalancen wie beim Einfügen). Die Höhenbalance des direkten Vorgängers muss ebenfalls angepasst werden.
- Fall 3: die Höhenbalance wurde auf $-2$ geändert und dessen linker Unterbaum hat eine Höhenbalance von 0 (der Knoten wurde aus dem rechten Unterbaum gelöscht) – dieser Fall kann nur beim Löschen auftreten. Es ist eine Rechtsrotation notwendig:




  Nach der Rotation hat der Unterbaum weiter die gleiche Höhe wie vorher. Die Höhenbalancen von B sind nun $+1$ und von D $-1$. Das Löschen ist damit beendet, es müssen keine weiteren Höhenbalancen angepasst werden.
- Fälle 4–6: sind symmetrisch zu den Fällen 1–3 – die Höhenbalance wurde auf $+2$ geändert, abhängig von der Höhenbalance des linken Unterbaums ist eine Links- oder Rechtslinksrotation notwendig. Ggf. muss die Höhenbalance des direkten Vorgängers angepasst werden.

Beim Löschen kann pro Ebene höchstens eine (Doppel-)Rotation notwendig sein. Also auch hier ist der Aufwand durch zwei mal Höhe des AVL-Baums beschränkt.

**Wie hoch kann ein AVL-Baum mit $n$ Knoten werden?** Es bleibt noch zu zeigen, dass der Aufwand zum Suchen, Einfügen und Löschen in AVL-Bäumen tatsächlich nur $O(\log n)$ beträgt – bisher haben wir lediglich gezeigt, dass sie höchstens proportional zur Höhe des AVL-Baums ist. Wir konstruieren nun einen AVL-Baum mit möglichst großer Höhe bei Verwendung

möglichst weniger Knoten. Hat ein AVL-Baum die Höhe $h$, so muss ein Unterbaum die Höhe $h-1$ haben und der andere – aufgrund der Beschränkung der Höhenbalancen – die Höhe $h-2$. Dieses Prinzip lässt sich rekursiv fortsetzen. Die Anzahl der Knoten in so einem Baum lässt sich ebenfalls leicht darstellen. Ein AVL-Baum der Höhe 1 hat $g(1) = 1$ Knoten, ein AVL-Baum der Höhe 2 hat mindestens $g(2) = 2$ Knoten, ein AVL-Baum der Höhe $h$ hat mindestens $g(h) = g(h-1) + g(h-2) + 1$ Knoten. Die Rekursionsformel hat nicht nur starke Ähnlichkeit mit der der Fibonacci-Zahlen ($f(1) = 1$, $f(2) = 2$, $f(h) = f(h-1) + f(h-2)$), es gilt: $g(h) = f(h+1) - 1$ (mit vollständiger Induktion zeigen wir $g(1) = f(2) - 1 = 1$ und $g(h) = g(h-1)+g(h-2)+1 = f(h)-1+f(h-1)-1+1 = f(h+1)-1$). Für Fibonacci-Zahlen gilt $f(h) \approx ((\sqrt{5}-1)/2)^{h+1}/\sqrt{5}$, damit ist auch $g(h) \in O(((\sqrt{5}-1)/2)^h)$. Mit $n \geq c \cdot ((\sqrt{5}-1)/2)^h$ gilt auch $h \in O(\log n)$ – bei genauerer Analyse lässt sich zeigen, dass $h \leq 1.4405 \cdot \log(n+2)$.

**Der C++-Code** arbeitet vom Prinzip her sehr ähnlich wie bei normalen Suchbäumen, muss aber zusätzlich die Höhenbalancen anpassen. Die Implementierungen, die im Netz verfügbar sind, sind quasi alle nicht effizient, da sie zusätzlich Zeiger auf die Vorgänger verwalten. Dies ist nicht notwendig. Der hier vorgestellte C++-Code ist zwar effizienter, die Wechselwirkungen von Rekursion und Referenzparametern sind aber nicht ganz leicht nachzuvollziehen, wir verzichten hier auf weitere Details, der Code ist im Anhang A abgedruckt.

### 1.3.9 B-Bäume

Mit AVL-Bäumen haben wir nun eine Datenstruktur kennengelernt, die auch im Worst-Case stets nur logarithmisch viele Schritte für das Suchen, Einfügen und Löschen von Elementen benötigt. In manchen Anwendungsfällen wird man jedoch trotzdem auf andere Datenstrukturen ausweichen. Dies ist z. B. dann der Fall, wenn die Daten zu umfangreich sind, um vollständig im Hauptspeicher des Rechners gehalten zu werden. Liegen die Daten auf externen Speichern, sind häufig die Zugriffszeiten deutlich größer als die Verarbeitungszeiten im PC – man wird hier versuchen, die Anzahl der Zugriffe auf die externen Speicher gering zu halten. Eine Möglichkeit dazu bieten B-Bäume (ebenfalls ein Suchbaum mit weiteren Eigenschaften). Wir verweisen hier lediglich auf die Literatur.

## 1.4 Hashing

Hashing ist eine gänzlich andere Art Suchalgorithmus. Hier werden die Daten in einer Tabelle abgelegt. Im Gegensatz zu Suchbäumen oder Intervallschachtelung wird hier die Position der Daten in der Tabelle berechnet. Im Mittel ist Hashing meist schneller als die Verwendung von Suchbäumen, die geringe Laufzeit kann (ähnlich wie bei der Interpolationssuche im Vergleich zur binären Suche) jedoch nicht garantiert werden. Auch hier verweisen wir lediglich auf die Literatur.

# 2 Sortieralgorithmen

Sortieren in C++ ist sehr einfach: man füge ein **#include**<algorithm> hinzu und rufe zum Sortieren der ersten 7 Elemente des Arrays a einfach den Befehl sort(a,a+7); auf.

Auch, wenn man in vielen Fällen mit dieser Routine der C++ Standard Library arbeiten kann, haben wir dadurch noch keinerlei Verständnis für das Sortierproblem entwickelt. So vielfältig

die Anwendungen sind, in denen Sortieralgorithmen benötigt werden, so vielfältig sind auch die verschiedenen Ansätze. Wir wollen einige dieser Ansätze hier vorstellen, die alle ihre Vor- und Nachteile haben. Den ultimativen Sortieralgorithmus für alle Gelegenheiten gibt es nicht und so ist es ratsam, sich dieser Eigenschaften bewusst zu sein. Darüberhinaus lassen sich verschiedene Programmiertechniken schön an diesen Algorithmen verdeutlichen.

Wir betreiben zunächst ein wenig Theorie und werden dann einfache und effiziente Sortieralgorithmen vorstellen.

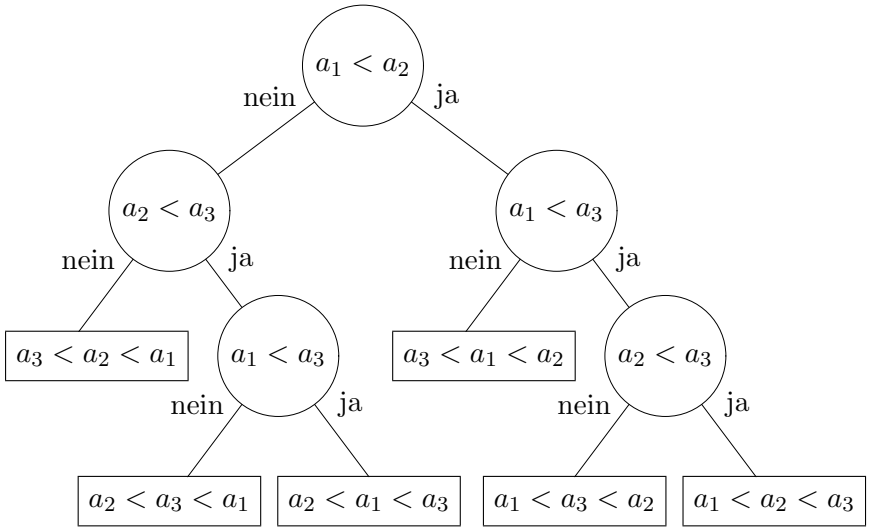
## 2.1 Ein wenig Theorie

**Aufgabe des Sortierens:** Gegeben ist ein Folge von Zahlen $a_1$, $a_2$, …, $a_n$; gesucht ist eine Umordnung dieser Zahlen, also eine Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, so dass $a_{\pi(1)}$, $a_{\pi(2)}$, …, $a_{\pi(n)}$ geordnet ist, d. h., es soll $a_{\pi(i)} < a_{\pi(i+1)}$ für $i \in \{1, \ldots, n-1\}$ gelten.

*Inversionszahl, Fehlstand*

**Ein Maß für die Unsortiertheit:** Die Anzahl der Paare von Zahlen (nicht notwendigerweise direkt benachbarte), die in der Folge $A$ ($a_1$, $a_2$, …, $a_n$) in falscher Reihenfolge stehen, heißt Inversionszahl (oder Fehlstand) $I(A) = |\{(i,j) \mid i < j \text{ und } a_i > a_j\}|$. Die Inversionszahl liegt dabei zwischen 0 (sortierte Folge) und $\frac{(n-1)n}{2}$ (umgekehrt sortierte Folge).

**Wie aufwendig ist Sortieren?** Da wir nicht wissen, welche Zahlen in falscher Reihenfolge stehen, werden wir Zahlen vergleichen. Betrachten wir das Sortieren von drei Zahlen $a_1$, $a_2$ und $a_3$ – in den (runden) Knoten stehen die durchgeführten Vergleiche, in den Rechtecken das Ergebnis des Sortierens:




*eine untere Schranke für das vergleichsbasierte Sortieren*

Es gibt $n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1 = n!$ Möglichkeiten, die $n$ Elemente einer Folge anzuordnen (im Beispiel 3!=6 Möglichkeiten die 3 Elemente anzuordnen) – gesucht ist genau die eine Möglichkeit, so dass die Elemente in aufsteigender Reihenfolge sind. An den Nullzeigern dieses Entscheidungsbaums stehen die sortierten Folgen, d. h., ein Entscheidungsbaum muss $n!$ Nullzeiger haben (jede der $n!$ Anordnungen muss eindeutig erkannt werden). Ein Binärbaum mit $n!$ Nullzeigern hat genau $n! - 1$ Knoten (im Beispiel 3!=6 Nullzeiger und 5 Knoten). Die Höhe

eines Binärbaumes mit $k$ Knoten beträgt mindestens $\lceil \log(k+1) \rceil$. Jeder Knoten im Entscheidungsbaum entspricht einem Vergleich, d. h., der Entscheidungsbaum zum Sortieren hat eine Höhe von mindestens $\lceil \log(n!) \rceil$ (im Beispiel war somit vorher bereits klar, dass die Höhe mindestens $\lceil \log(3!) \rceil = 3$ betragen musste, obiger Baum ist diesbzgl. also optimal). Mit Hilfe der Stirlingschen Formel folgt: Jedes Sortierverfahren, das ausschließlich auf dem Vergleichen von Zahlen basiert benötigt im Worst-Case mindestens $\lceil \log(n!) \rceil \approx n \log n - 1.4404 \cdot n$ Vergleiche.

Wir werden natürlich nicht für jedes $n$ einen optimalen Entscheidungsbaum entwerfen (das zugehörige Programm hätte ja $n! - 1$ **if**-Abfragen und wäre viel zu groß), sondern versuchen, die Entscheidungen, wie die Elemente umzusortieren sind, nach einem festen strukturierten Schema durchzuführen. Es ist lediglich sicher, dass – vergleichsbasiert – kein Verfahren unter der Laufzeit von $O(n \log n)$ liegen kann.

## 2.2 Einfache Sortieralgorithmen

*Minimumsort*

Ein einfaches Sortierverfahren – Sortieren durch Minimumsuche – haben wir bereits im ersten Semester kennengelernt, ebenso haben wir dessen Aufwand bereits mit $O(n^2)$ abgeschätzt.

`sortieren.cpp`

```
void tausche(int & a, int & b) {
  int h=a; a=b; b=h;
}

void MinimumSort(int tosort[], const int l, const int r) {
  int mindex;
  for(int i=l;i<r;i++) {
    mindex=i;
    for(int j=i+1;j<=r;j++) { if (tosort[j]<tosort[mindex]) { mindex=j; } }
    tausche(tosort[i],tosort[mindex]);
  }
}
```

*Bubblesort*

Ein weiteres weit verbreitetes Sortierverfahren ist Bubblesort, das auf dem Austausch von benachbarten Elementen beruht. Hierbei wird nach dem ersten Durchlauf der äußeren Schleife garantiert, dass das größte Element ganz ans Ende des Feldes „gebubblet“ wurde. Nach dem $i$-ten Durchlauf der äußeren Schleife sind die letzten $i$ Elemente garantiert sortiert. Ohne weitere Maßnahmen können wir die Anzahl der benötigten Vergleiche exakt angeben: $(n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1)n}{2}$, der Aufwand beträgt also $O(n^2)$. Wenn in einem Durchlauf kein Vertauschen mehr nötig war, kann man das Verfahren auch frühzeitig abbrechen. Bauen Sie diese Verbesserung ein und ermitteln Sie experimentell, wie groß die Verbesserung ist (Vorwarnung: seien Sie nicht enttäuscht).

Übung

`sortieren.cpp` (Fortsetzung)

```
void BubbleSort(int tosort[], const int l, const int r) {
  for(int i=l;i<r;i++) {
    for(int j=l;j<r-(i-l);j++) {
      if (tosort[j]>tosort[j+1]) { tausche(tosort[j],tosort[j+1]); }
    }
  }
}
```

Leider gibt es Beispiele mit relativ geringem Fehlstand, bei denen Bubblesort auch mit dem vorzeitigen Abbruchkriterium, quadratische Laufzeit benötigen (betrachten Sie die aufsteigend sortierte Folge bei der lediglich die 1 ans Ende der Folge verschoben wurde). Übung

Insertionsort ist gegenüber Bubblesort optisch wenig verändert. Das Verfahren nimmt jeweils das nächste Element und lässt es soweit nach links „bubblen“, bis es (bzgl. der bisher betrachteten Elemente) an der richtigen Stelle steht. Nach dem $i$-ten Durchlauf der äußeren Schleife sind die ersten $i$ Elemente garantiert sortiert. *Insertionsort*

`sortieren.cpp` (Fortsetzung)

```
void InsertionSort(int tosort[], const int l, const int r) {
  for(int i=l+1;i<=r;i++) {
    for(int j=i;j>l;j--) {
      if (tosort[j]<tosort[j-1]) { // dann tauschen
        tausche(tosort[j-1],tosort[j]);
      } else break; // sonst Schleifenabbruch
    }
  }
}
```

Trotz dieser Verwandschaft hat Insertionsort gegenüber Bubblesort einen großen Vorteil (der Bubblesort in jeglicher Hinsicht überflüssig macht): Mit jedem Vergleich wird entweder danach das Paar vertauscht und der Fehlstand sinkt um 1 oder das gerade einsortierte Element hat seine korrekte Position gefunden. Jedes Element findet genau einmal seine korrekte Position (Aufwand in der Summe $O(n)$), damit lässt sich der Aufwand von Insertionsort (in der hier vorliegenden Fassung) mit $O(n+I(A))$ angeben, der Aufwand hängt also direkt davon ab, wie unsortiert die Zahlenfolge ist. Insbesondere benötigt es nie mehr Vergleiche als (die verbesserte Version von) Bubblesort.

**Wichtig:** Die Laufzeit ist nur so gut, wenn man das neue Element von rechts nach links einsinken lässt. Geht man andersherum vor und sucht jeweils von links nach rechts die richtige Position (und verschiebt die schon sortierten Elemente ein Feld nach rechts), so benötigt Insertionsort stets $O(n^2)$ Schritte. Prüfen Sie diese Behauptung an einigen Beispielen von Hand nach und vergleichen Sie die notwendigen Schritte mit der oben vorgestellten besseren Variante von Insertionsort und mit Bubblesort. Übung

Die einfachen Sortierverfahren benötigen im Worst-Case stets $O(n^2)$ Schritte. Die Lücke zur unteren Schranke $\approx n \log n$ ist groß. Für die Praxis hat nur das Insertionsort eine Daseinsberechtigung: haben sich nur $k$ Elemente verändert und soll die gesamte Folge neu sortiert werden, so beträgt der Fehlstand $O(k \cdot n)$ und damit auch die Laufzeit zum (erneuten) Sortieren lediglich $O(k \cdot n)$.

## 2.3 Effiziente Sortieralgorithmen

Mit Hilfe derAVL-Bäume haben wir ganz nebenbei auch die Grundlage zu einem effizienten Sortierverfahren gewonnen (ein schönes Beispiel zur Wiederverwendung von bestehendem Code): da das Einfügen nur $O(\log n)$ Schritte benötigt und die (sortierte) Inorder-Ausgabe in Linearzeit möglich ist, beträgt die Gesamtlaufzeit für das Sortierverfahren $O(n \log n)$, es ist also bis auf konstante Faktoren optimal. Diese Optimalität (und auch Flexibilität, denn wir können jeweils in logarithmischer Zeit Elemente löschen, hinzufügen oder auch ändern (wie?)) wird erkauft durch zusätzlichen Speicherbedarf: pro Element braucht man zwei Zeiger auf dessen Kinder. *Sortieren mit AVL-Bäumen*

Die Frage ist also, ob dies besser geht. Wir stellen hier drei weitere Verfahren vor.

*Quicksort*

Eines der meistverwendeten Sortierverfahren ist Quicksort. In der Literatur finden sich viele verschiedene Varianten, allen gemein ist dabei die Grundidee, die Elemente in kleinere und größere Elemente aufzuteilen (wobei das größte der kleinen nicht größer als das kleinste der großen ist) und dann diese beiden Teilmengen rekursiv (mit demselben Verfahren) unabhängig voneinander zu sortieren. Diese Idee, ein Problem in mehrere kleinere Probleme derselben Art aufzuteilen und die Teillösungen dann zu einer Gesamtlösung zusammenzufügen, ist ein allgemeiner Lösungsansatz und nennt sich Teile-und-beherrsche (oder kurz Teile-und-herrsche, engl. Divide-and-Conquer). Ein Beispiel für diesen Lösungsansatz haben wir bereits im ersten Semester mit den Türmen von Hanoi kennengelernt. Er führt (fast immer) zu rekursiven Ansätzen und dementsprechend auch bei den Laufzeitabschätzungen zu Rekursionsgleichungen. Glücklicherweise sind diese oft leicht aufzulösen.

*Teile-und-(be)herrsche, Divide-and-Conquer*

Betrachten wir nun den Quicksort-Algorithmus ausführlich an einem konkreten Beispiel:

| 58 | 97 | 93 | 23 | 84 | 62 | 64 | 33 | 83 | 27 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|

Das erste Problem ist, wir wissen nicht, welches Element das „größte der kleineren“ sein wird – wir müssen raten, welches Element dies ist. Zum Aufteilen der Folge bzgl. einem Element nehmen wir hier das mittlere Element der Folge (im Array steht dies am Index (l+r)/2, wenn l der Index des linkesten Elements und r der des rechtesten sind). Dieses Element (bzgl. dessen Wert die Folge in kleinere und größere Elemente aufgeteilt wird) nennt man Pivot-Element. Im ersten Schritt ist dies die 62. Wir werden nun vom linken und rechten Rand aus, Elemente suchen, die auf der falschen Seite stehen (also Elemente links, die größer als 62 sind, und solche rechts, die kleiner als 62 sind). Ausgehend von

*Pivot-Element*

| 58 | 97 | 93 | 23 | 84 | 62 | 64 | 33 | 83 | 27 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\uparrow_L$ | | | | | | | | | | $\uparrow_R$ |

wandert der linke Zeiger bis zur 97 und der rechte bis zur 27.

| 58 | 97 | 93 | 23 | 84 | 62 | 64 | 33 | 83 | 27 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\uparrow_L$ | | | | | | | | $\uparrow_R$ | |

Nach Vertauschen dieser beiden Elemente wissen wir, dass die ersten 2 Elemente kleiner als 62 sind und die letzten beiden größer als 62. Bei der Suche nach falsch stehenden Elementen können die Zeiger jeweils eine Position weitergerückt werden.

| 58 | 27 | 93 | 23 | 84 | 62 | 64 | 33 | 83 | 97 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $\uparrow_L$ | | | | | | $\uparrow_R$ | | |

Die Zeiger bleiben nun bei 93 und 33 stehen, die beiden Werte werden getauscht, die Zeiger rücken jeweils eine Position weiter:

| 58 | 27 | 33 | 23 | 84 | 62 | 64 | 93 | 83 | 97 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\uparrow_L$ | | | $\uparrow_R$ | | | | |

Die Zeiger wandern wieder jeweils weiter, der linke bleibt bei der 84 stehen, der rechte Zeiger wandert bis zur 62 (nicht, weil es die Mitte ist, sondern weil ein Element gefunden wurde das kleiner oder gleich dem Pivot-Element ist – der Grund, warum wir auch bei Gleichheit den

Zeiger nicht weiterrücken, wird am Ende des Quicksort-Algorithmus nachgereicht). Wieder werden Elemente getauscht und die Zeiger eine Position weitergerückt.

| 58 | 27 | 33 | 23 | 62 | 84 | 64 | 93 | 83 | 97 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | $\uparrow_R$ | $\uparrow_L$ | | | | | |

Die Zeiger sind nun aneinander vorbei gelaufen, daher findet kein Tausch mehr statt (wir wollen ja zuweit links stehende Elemente mit zuweit rechts stehenden vertauschen – weitere Vertauschungen würden jetzt in die entgegengesetzte Richtung laufen).

Wir wissen nun, dass alle Elemente vom linken Rand bis zum rechten Zeiger kleiner gleich der 62 sind und alle vom linken Zeiger bis zum rechten Rand größer gleich 62. Die entsprechenden Teilfelder werden nun rekursiv mit dem Quicksort-Verfahren sortiert. Zunächst das Teilfeld von 58 bis 62: als Pivot-Element wird die 33 gewählt, der linke Zeiger bleibt bei der 58, der rechte wandert zur 23, die beiden Elemente werden getauscht; danach wandern der linke und rechte Zeiger jeweils bis zur 33, die Zeiger können jeweils eins weitergerückt werden; die Zeiger sind aneinander vorbei gelaufen, es findet kein Tausch mehr statt; die rekursiv zu sortierenden Teilfelder sind die vom linken Rand bis zum rechten Zeiger (23 bis 27) und vom linken Zeiger bis zum rechten Rand (58 bis 62) – die 33 ist bei keinem der beiden Teilfelder dabei.

| 58 | 27 | 33 | 23 | 62 | 84 | 64 | 93 | 83 | 97 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\uparrow_L$ | | | | $\uparrow_R$ | | | | | | |
| 23 | 27 | 33 | 58 | 62 | 84 | 64 | 93 | 83 | 97 | 95 |
| | $\uparrow_L$ | $\uparrow_R$ | | | | | | | | |
| 23 | 27 | 33 | 58 | 62 | 84 | 64 | 93 | 83 | 97 | 95 |
| | $\uparrow_R$ | | $\uparrow_L$ | | | | | | | |

Betrachten wir wieder das linke Teilfeld zuerst (von 23 bis 27): das Pivot-Element ist die 23, der linke Zeiger bleibt auf der 23, der rechte wandert bis zur 23, die Zeiger können jeweils eins weitergerückt werden (der rechte Zeiger rückt dabei über den linken Rand hinaus, der linke auf die 27); die rekursiv zu sortierenden Teilfelder sind die vom linken Rand bis zum rechten Zeiger (dieses Teilfeld ist leer, es ist nichts zu tun) und vom linken Zeiger bis zum rechten Rand (nur die 27, auch hier ist nichts mehr zu tun). Beim anderen Teilfeld (von 58 bis 62) sind die Schritte analog, auch hier ist bei den dann auftretenden rekursiven Aufrufen nichts mehr zu tun.

Kommen wir zurück zum noch zu sortierenden Teilfeld von 84 bis 95. Das Pivot-Element ist die 93 der linke Zeiger wandert bis zur 93, der rechte zur 83. Nach dem Tausch sind die Zeiger aneinander vorbeigelaufen, die rekursiv zu sortierenden Teilfelder sind die vom linken Rand bis zum rechten Zeiger (84 bis 83) und vom linken Zeiger bis zum rechten Rand (93 bis 95).

| 23 | 27 | 33 | 58 | 62 | 84 | 64 | 93 | 83 | 97 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | $\uparrow_L$ | | | | | $\uparrow_R$ |
| 23 | 27 | 33 | 58 | 62 | 84 | 64 | 83 | 93 | 97 | 95 |
| | | | | | | | $\uparrow_R$ | $\uparrow_L$ | | |

Im Teilfeld 84 bis 83 wird als Pivot-Element zufällig das kleinste Element, die 64, gewählt. Der linke Zeiger bleibt auf der 84, der rechte wandert zur 64. Nach dem Tausch sind die Zeiger aneinander vorbeigelaufen, die rekursiv zu sortierenden Teilfelder sind die vom linken Rand

bis zum rechten Zeiger (nur die 64) und vom linken Zeiger bis zum rechten Rand (84 bis 83).

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 27 | 33 | 58 | 62 | 84 | 64 | 83 | 93 | 97 | 95 |
| | | | | | $\uparrow_L$ | | $\uparrow_R$ | | | |
| 23 | 27 | 33 | 58 | 62 | 64 | 84 | 83 | 93 | 97 | 95 |
| | | | | | $\uparrow_R$ | $\uparrow_L$ | | | | |

Im einelementigen Teilfeld 64 ist nichts zu tun, im Teilfeld 84 bis 83 bleiben die Zeiger jeweils auf 84 bzw. 83 stehen, nach dem Tausch finden rekursive Aufrufe für die dann einelementigen Teilfelder 83 und 84 statt. Es verbleibt das Teilfeld 93 bis 95, die Schritte sind analog zu obigen Teilfeldern. Danach ist das gesamte Feld sortiert.

`sortieren.cpp` (Fortsetzung)

```
void QuickSort(int tosort[], const int l, const int r) {
  if (l<r) {
    int pl=l;
    int pr=r;
    int pivot=tosort[(l+r)/2];
    while (pl<=pr) {
      while (tosort[pl]<pivot) { pl++; }
      while (pivot<tosort[pr]) { pr--; }
      if (pl<=pr) { tausche(tosort[pl],tosort[pr]); pl++; pr--; }
    }
    QuickSort(tosort,l,pr);
    QuickSort(tosort,pl,r);
  }
}
```

Auch, wenn der Ablauf kompliziert erscheint, führt das Programm genau diese Schritte aus. Will man sich die wiederholte Übergabe des Arrays mit jedem rekursiven Aufruf sparen, so kann dieser Zugriff auch über eine (nur im Modul `sortieren.cpp` bekannten) Zeigervariable geschehen (die Funktionalität des Quicksort würde dann in eine nur modulweit bekannte Hilfsfunktion verschoben, der Aufruf von Quicksort würde dann lediglich den Zeiger auf das Array in die globale Variable geschrieben und die Hilfsfunktion mit den Indizes des linken und rechten Rands aufgerufen[18]).

Auf einige scheinbare Details des Algorithmus müssen wir hier noch eingehen (diese verdeutlichen auch, wie schwierig es letzten Endes ist, alle möglichen Randfälle mit zu beachten):

In manchen Fällen wird das Pivot-Element von seiner Position weggetauscht und die Zeiger sind noch nicht übereinander hinweg gelaufen – in diesem Fall bleibt das Pivot-Element das ursprünglich gewählte (der Wert ist entscheidend, nicht wo dieser Wert steht). Finden Sie ein Beispiel, bei dem dieses eine Rolle spielt. Übung

In Zeile 9 wird auch bei Gleichheit der beiden Zeiger getauscht! Diese Situation trat oben auch auf, als das Teilfeld 58 bis 62 betrachtet wurde und beide Zeiger bis zur 33 wanderten. Das Weitersetzen der Zeiger ist in diesem Fall sinnvoll und nötig. Da der Fall der Gleichheit relativ selten ist, wäre der Aufwand, den unnötigen Tausch zu vermeiden, größer als die zu erwartende Ersparnis.

Als letztes Detail fällt vielleicht auf, dass die **while**-Schleife auch dann betreten wird, wenn beide Zeiger am selben Index stehen. Ist dieses Element in dem Moment das Pivot-Element,

[18] Im gleich beschriebenen Sortierverfahren Mergesort werden wir dies so ausführen.

so spielt es keine Rolle, ob wir das Element zur linken oder rechten Teilmenge hinzufügen oder es – wie in obiger Situation der 33 – garnicht in den rekursiven Teilfeldern berücksichtigen. Es gibt jedoch Fälle, bei denen die Zeiger zu Beginn der **while**-Schleife auf dem selben Index stehen, dieses aber nicht das Pivot-Element ist. In diesem Fall müssen wir entscheiden, ob der linke Zeiger noch eine Position nach rechts oder der rechte Zeiger eine Position nach links muss. Auch hier ist es so, dass der Zusatzaufwand zur Erkennung dieser speziellen Situation höher wäre, als – wie hier geschehen – die Frage durch einen weiteren Durchlauf der **while**-Schleife geklärt wird. Betrachten Sie hierzu die Folge (2,3,4,5,7,9,8,6,1) – der Verbleib der „8“ bliebe ungeklärt. Übung

Bleibt noch die Frage zu klären, warum bei Gleichheit von Element und Pivot-Element die Zeiger nicht weiterrücken, obwohl das Element auch an der betreffenden Position an der richtigen Stelle steht. Betrachten Sie dazu die Folge (6,6,6), Pivot-Element ist die 6; rückt man bei Gleichheit den Zeiger weiter, rutscht der linke Zeiger (ohne dass das Programm dies merkt) über den rechten Rand des Feldes hinaus und verursacht so einen unerlaubten Speicherzugriff. Auch hier sind ständige zusätzliche Indexabfragen aufwendiger als bei einigen gleichen Elementen, diese ggf. untereinander zu vertauschen.

**Wie aufwendig ist Quicksort?** Wenn man sich auf einfache Laufzeittests beschränkt, stellt man fest, dass Quicksort im Mittel etwa $1.386 \cdot n \log n$ Vergleiche durchführt. In $O$-Notation brauchen wir jedoch eine obere Schranke für den schlechtest-möglichen Fall. Stellen wir uns vor, wir sollen die Zahlen 1 bis $n$ sortieren und die 1 würde als Pivot-Element ausgewählt. Nach $n$ Vergleichen ist das Teilfeld mit den Elementen 2 bis $n$ erkannt. In diesem könnte wiederum die 2 als Pivot-Element gewählt werden, so dass $n-1$ Vergleiche zum Aufteilen in die 2 und die Teilmenge 3 bis $n$ benötigt werden. So erhält man einen Aufwand von $n + (n-1) + \cdots \in O(n^2)$. So kann Quicksort zum „Slowsort“ werden. Konstruieren Sie für $n = 9$ ein Feld, bei dem in jedem rekursiven Aufruf das kleinste oder größte Element als Pivot-Element gewählt wird. Übung

Quicksort erfüllt somit zwar die Anforderung mit weniger Speicherplatz auszukommen[19], kann die schnelle Laufzeit aber nicht garantieren.

Wir schlagen hier als Blick über den Stoff des zweiten Semesters hinaus nochmal einen Bogen zu den binären Suchbäumen und betrachten auch hier eine theoretische Herleitung für den Aufwand zum Sortieren im Mittel (Anzahl der Element-Vergleiche). Wir gehen dazu davon aus, dass wir das Pivot-Element zufällig wählen (Effekte, die durch vorsortierte Felder auftreten, werden hier also nicht berücksichtigt). Wir können mit $n$ Vergleichen den Quicksort-Schritt durchführen und teilen dabei das Feld in zwei nicht-leere Teilfelder mit $i$ und $n-i$ Elementen auf, wobei durch das Verfahren $i > 0$ und $i < n$ sichergestellt sind. Durch die angenommene Zufälligkeit der Wahl des Pivot-Elements bilden wir hier das Mittel der $n-1$ möglichen Aufteilungen in zwei Teilfelder. Dies führt auf die Rekursionsgleichung

$$Q(n) = n + \frac{1}{n-1} \cdot \sum_{i=1}^{n-1} (Q(i) + Q(n-i))$$

[19] Lediglich die Rekursion benötigt hier zusätzlichen Speicherplatz. Wenn das kleinere Teilfeld jeweils zuerst betrachtet wird, ist dieser Platzbedarf logarithmisch beschränkt – vorausgesetzt, der Compiler erkennt, dass der andere rekursive Aufruf der letzte Befehl ist und die lokalen Variablen auf dem programminternen Stack zusammen mit diesem Aufruf aufgeräumt werden (Stichwort Tail-Rekursion). Ansonsten kann der zusätzlich benötigte Speicher auf dem Stack linear sein. Will man sicher gehen, dass lediglich logarithmischer Zusatzspeicher benötigt wird, kann man die Rekursion auch selbst auf einem eigenen Stack verwalten (Details selbst überlegen).

Hierbei kommt jeder Summand der $Q(i)$ genau zwei Mal vor. Wir erhalten somit (die zweite Gleichung entsteht durch Ersetzen von $n$ mit $n-1$):

$$Q(n) = n + \frac{2}{n-1} \cdot \sum_{i=1}^{n-1} Q(i) \qquad (③)$$

$$Q(n-1) = n - 1 + \frac{2}{n-2} \cdot \sum_{i=1}^{n-2} Q(i) \qquad (④)$$

Die unangenehme Summe fällt durch geschicktes Bilden der Differenz $(n-1) \cdot ③ - (n-2) \cdot ④$ bis auf das letzte Glied vollständig heraus.

$$(n-1) \cdot Q(n) - (n-2) \cdot Q(n-1) = n(n-1) - (n-1)(n-2) + 2 \cdot Q(n-1)$$

Durch Zusammenfassen der Terme erhalten wir

$$(n-1) \cdot Q(n) = 2(n-1) + n \cdot Q(n-1)$$

Um die Terme mit $Q(n)$ und $Q(n-1)$ auf beiden Seiten gleichartiger dargestellt zu haben, teilen wir beide Seiten durch $n(n-1)$

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

Wir können die Gleichung nun rekursiv einsetzen und erhalten mit $Q(0) = 0$, $Q(1) = 0$ und der Harmonischen Reihe $H(n) = \sum_{i=1}^{n} \frac{1}{i}$ (in der betrachteten Summe fehlt durch $Q(1) = 0$ das erste Glied, wir müssen daher $2 \cdot \frac{1}{1}$ wieder abziehen)

$$\frac{Q(n)}{n} = 2 \cdot H(n) - 2$$

Mit der Näherung $H(n) \approx \ln n + \gamma$ ($\gamma \approx 0.577$ ist die Eulersche Konstante) erhalten wir für die zu erwartende Anzahl der Element-Vergleiche (Index-Vergleiche wurden nicht berücksichtigt) bei Quicksort

$$Q(n) \approx 1.386 n \log n - 0.846n$$

Überprüfen Sie experimentell, wie gut dieser Wert in der Praxis stimmt – wir wählen als Pivot-Element dafür wie gehabt das Element, das im zu betrachtenden Teilfeld in der Mitte steht. Übung

Die Abschätzung des Aufwands ist nicht zufällig sehr ähnlich wie bei der mittleren Suchdauer in binären Suchbäumen. Letzten Endes arbeiten wir bei Quicksort wie bei binären Suchbäumen: wir wählen zufällig ein Pivot-Element (Wurzel des Suchbaums) und bearbeiten die kleineren Elemente und die größeren Elemente rekursiv mit demselben Verfahren.

Ein weiteres Verfahren, das ebenfalls den Divide-and-Conquer-Ansatz verfolgt und garantiert lediglich $n \log n$ Vergleiche benötigt, ist Mergesort. Die Idee ist dabei verblüffend einfach: teile das Feld in zwei gleichgroße Teile, sortiere rekursiv jedes Teilfeld mit Mergesort und mische die beiden Teilfelder. Einziger Haken an dem Verfahren ist der auch hier notwendige lineare Zusatzspeicher: es wird Platz benötigt, um zumindest die Hälfte der Elemente zwischenspeichern zu können. *Mergesort*

Das Mischen von zwei sortierten Arrays ist einfach: das kleinste Element ist das erste vom einen oder das erste vom anderen Feld; das jeweils nächste Element im zusammengefügten

Array ist stets eines der beiden ersten noch nicht verwendeten Elemente. Ein Beispiel: beim Mischen von (31,41,59) und (26,53,58) wird das erste Element 26 durch den Vergleich $31 < 26$ gefunden, der Vergleich $31 < 53$ liefert die 31, der Vergleich $41 < 53$ die 41, dann $59 < 53$ die 53 und schließlich $59 < 58$ die 58. Ist eines der Arrays zu Ende, folgen die restlichen Elemente des anderes Arrays am Ende (hier nur noch das Element 59).

Der Aufwand lässt sich leicht rekursiv beschreiben: bei nur einem Element benötigt man $t(1) = 1$ Schritt, ansonsten sortiert man 2 Teilfelder mit je $n/2$ Elementen und benötigt noch $n$ Schritte zum Mischen, somit $t(n) = 2 \cdot t(n/2) + n$. Für $n = 2^k$ zeigen wir mit Induktion $t(2^k) = 2^k \cdot (k+1)$: es gilt

$$t(2^0) = t(1) = 1 = 2^0 \cdot 1 \text{ und } t(2^k) = 2 \cdot t(2^{k-1}) + 2^k = 2 \cdot 2^{k-1} \cdot k + 2^k = 2^k \cdot (k+1)$$

und somit $t(n) \in O(n \log n)$.

`sortieren.cpp` (Fortsetzung)

```
int *kopie; // Array für Kopie in MergeSort
int *mtosort; // modulweit bekanntes Array, zum Sparen der Parameterübergabe

void MergeSortRek(const int, const int); // Vorwärtsdeklaration

void MergeSort(int tosort[], const int l, const int r) {
  mtosort=tosort; // modulweit bekanntes Array initialisieren
  kopie=new int[(r-l)/2+1]; // maximal benötigter Zusatzspeicher,
  // wird so nur einmal angefordert, statt bei jedem Rekursionsaufruf erneut
  MergeSortRek(l,r);
  delete kopie;
}

void MergeSortRek(const int l, const int r) { // sortiert tosort[l..r]
  if (l<r) { // sonst ist nichts mehr zu tun
    int m=(l+r)/2;
    MergeSortRek(l,m);
    MergeSortRek(m+1,r);
    int lae1=m-l+1;
    for (int i=l;i<=m;i++) kopie[i-l]=mtosort[i];
    int p1=0; int p2=m+1; int pg=l; // linker Rand vom 1./2. Teilfeld
    while (p1<lae1 && p2<=r) { // solange noch Elemente vom 1. Teilfeld
      // in das 2. Teilfeld eingefügt werden müssen
      if (kopie[p1]<mtosort[p2]) {
        mtosort[pg++]=kopie[p1++];
      } else {
        mtosort[pg++]=mtosort[p2++];
      }
    }
    while (p1<lae1) { // ggf. restliche Elemente vom 1. TF ans Ende kopieren
      mtosort[pg++]=kopie[p1++];
    } // bleiben noch Elemente vom 2. TF übrig, stehen diese bereits richtig
  }
}
```

Die Frage ist, ob wir diese schnelle Laufzeit auch ohne den linear zusätzlichen Speicherplatz garantieren können? Wir können: Heapsort, hier in der verbesserten Variante Bottom-Up-Heapsort, garantiert den Aufwand $O(n \log n)$ mit nur konstant zusätzlichem Speicherbedarf.

*Bottom-Up-Heapsort*

Die Idee basiert auf dem einfachen Sortierverfahren Minimumsort: finde jeweils das kleinste Element der verbleibenden Daten und tausche es an die jeweils nächste Position. Die Verbesserung besteht darin, dass die Daten in einem Preprocessing-Schritt so verwaltet werden, dass in der Folge das jeweils kleinste Element schnell gefunden (und entfernt) werden. Wir stellen zunächst diese Datenstruktur, den Binär-Heap, vor und zeigen dann, wie sich dieser in einem effizienten Sortierverfahren verwenden lässt.

*Binär-Heap*

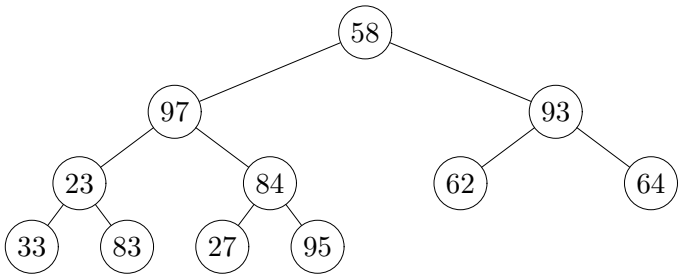
Wie lässt sich die Information „kleinstes Element“ so verwalten, dass gleichzeitig die nachfolgenden Elemente schnell gefunden werden können. Ein vollständig sortiertes Feld kann es sicher nicht sein, denn das ist ja gerade das Problem, das wir damit lösen wollen. Wir können aber eine Idee der Suchbäume übertragen: dort waren im linken Unterbaum alle kleineren und im rechten Unterbaum alle größeren Elemente. In einem (Minimum-)Heap sind in beiden Unterbäumen nur größere Elemente (ein Heap ist insbesondere kein Suchbaum). Dazu reicht es, die Eigenschaft aufrecht erhalten, dass der Wert jedes Knotens kleiner (oder gleich) dem seiner beiden Kinder ist (sofern er welche hat). Wenn dies gilt, so steht in der Wurzel des Heaps das Minimum aller Werte. In einem Maximum-Heap steht jeweils das größte Element oben. Aus programmiertechnischen Gründen fordern wir noch, dass der Baum ausgeglichen sein soll und zusätzlich alle Knoten in der untersten Ebene möglichst weit links stehen (man spricht hier meist von vollständig ausgeglichenen Binärbäumen). Betrachten wir wieder unsere Zahlen aus dem Quicksort-Beispiel

*(Minimum-, Maximum-) Heap*

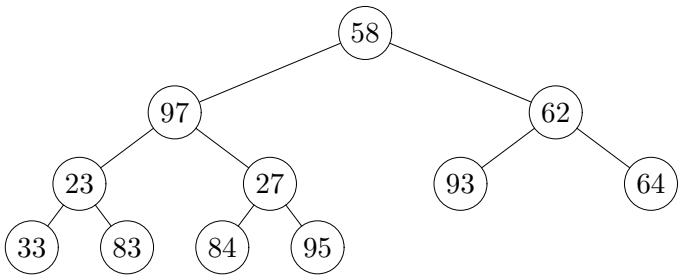
*vollständig ausgeglichener Binärbaum*

58 97 93 23 84 62 64 33 83 27 95

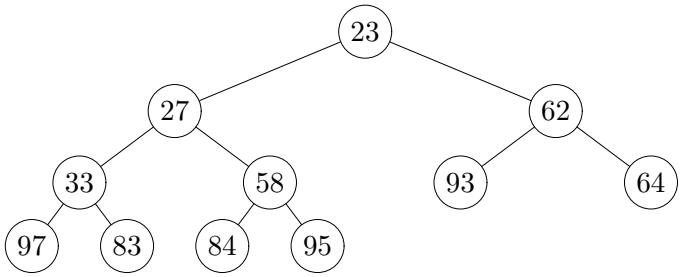
und tragen diese levelweise in einen vollständig ausgeglichenen Binärbaum ein:




Wir stellen die Heapeigenschaft jeweils durch „Reparatur“ an der Wurzel von Unterbäumen her, beginnend mit den kleinsten Unterbäumen, von „rechts nach links“. Die Blätter des Heaps erfüllen bereits die Heapeigenschaft, da sie keine Kinder haben. Die 84 erfüllt diese Eigenschaft nicht, muss also weiter unten im Unterbaum stehen. Daher muss die 84 mit dem kleineren der 2 Kindern vertauscht werden. Analog wird die 93 mit der 62 vertauscht (die 23 ist bereits kleiner als beide Kinder).

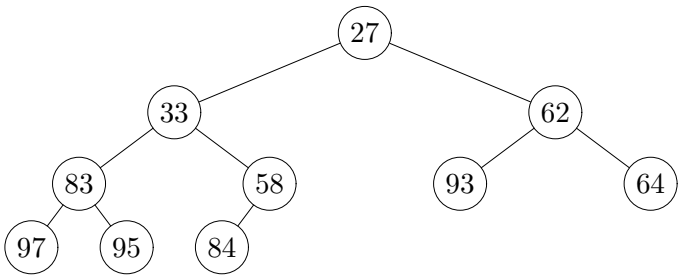


Nach Vertauschen der 97 mit der 23, muss die 97 danach noch weiter absinken (Tausch mit der 33). Nun verletzt nur noch die Wurzel die Heap-Eigenschaft: Die 58 wird zunächst mit der 23 und danach mit der 33 vertauscht. Nun ist die Heapeigenschaft erfüllt.

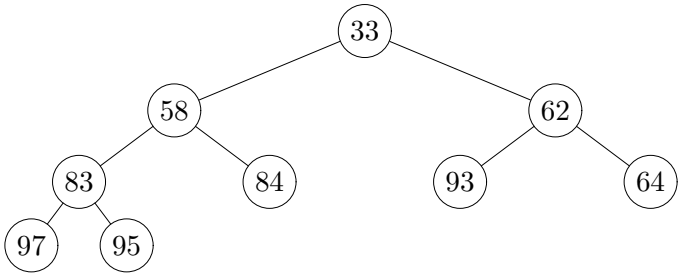



Die 23, das kleinste Element, steht nun in der Wurzel. Wird es entfernt, muss die Lücke mit der 95 aufgefüllt werden (ein Heap ist immer ein vollständig ausgeglichener Binärbaum); dadurch wird die Heapbedingung verletzt – jedoch nur an der Wurzel.

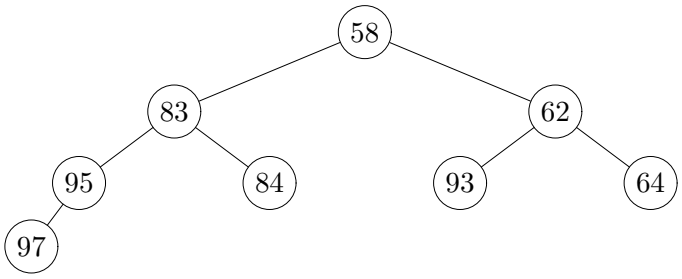
Die 95 muss also wieder im Heap an die richtige Position absinken: dies tut sie entlang der jeweils kleineren Kinder 23, 27, 33 und 83, die jeweils eine Ebene nach oben rücken.



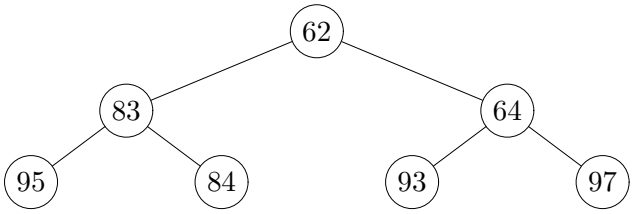

Nach Entfernen der 27 sinkt die 84 entlang der 33 und 58.



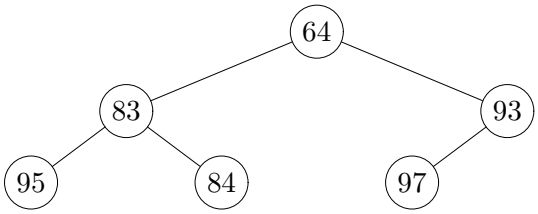

Danach wird die 33 entfernt und die 95 sinkt entlang der 58, 83. Der Knoten bleibt oberhalb des einzigen Kindes 97.



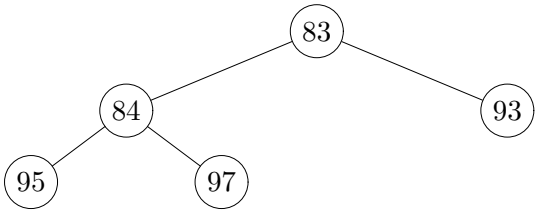

Danach wird die 58 entfernt und die 97 sinkt entlang der 62 und 64.



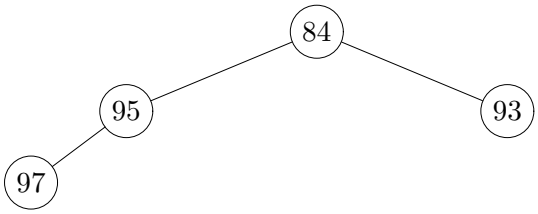
Danach wird die 62 entfernt und die 97 sinkt entlang der 64 und 93.



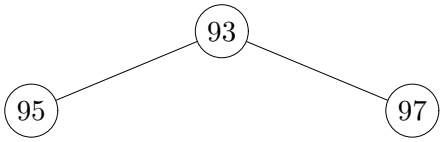

Danach wird die 64 entfernt und die 97 sinkt entlang der 83 und 84.




Danach wird die 83 entfernt und die 97 sinkt entlang der 84 und 95.




Danach wird die 84 entfernt und die 97 sinkt unter die 93.




Schließlich wird noch die 93 entfernt. Die 97 sinkt nochmals unter die 95. Die 95 wird entfernt und zuletzt die 97.

Beim Absinkenlassen benötigen wir im ersten Ansatz 2 Vergleiche pro Ebene: wir müssen die beiden Kinder vergleichen und, wenn der Knoten größer als das kleinere Kind ist, wird er mit diesem vertauscht. In der Praxis beobachtet man, dass die Elemente meist wieder in die letzte Ebene absinken (dies war meist auch in diesem Beispiel der Fall) – etwa 85% der Elemente sinken bis zur letzten Ebene, sogar über 98% sinken auf die letzte oder vorletzte Ebene. Der Pfad, auf dem ein Element absinken wird, kann vorab – und unabhängig von dem einsinkenden Element – mit einem Vergleich pro Ebene ermittelt werden! Man bestimmt also das Blatt, zu dem ein Element absinken würde (der Pfad, auf dem dies geschieht, heißt Einsinkpfad) – wenn man dort feststellt, dass der Elter größer ist, muss das Element wieder (mit einem Vergleich pro Ebene) aufsteigen.

*Einsinkpfad*

Binärbäume haben wir bisher mit Zeigern realisiert, damit wäre jedoch (wie bei Mergesort) linear viel zusätzlicher Speicherplatz nötig. Wie können wir uns also nun den Speicherplatz für die Zeiger sparen? Schreiben wir uns die Elemente des Heaps in Levelorder (also ebenenweise von links nach rechts) in ein Array, so stellen wir fest, dass es eine einfache Beziehung zwischen den Indizes eines Elter und seiner Kinder gibt: Beginnt man bei der Zählung der Indizes mit 1 (die Wurzel des Heaps), so hat ein Elter mit Index $i$ die Kinder mit Indizes $2i$ und $2i + 1$ (damit die Zählung der Indizes mit 1 beginnen kann, werden wir im Programmcode mit den Zeigern etwas tricksen). Wir können also leicht vom Index des Elter auf die Indizes der Kinder schließen. Von einem Knoten auf den direkten Vorgänger ist es noch einfacher: man halbiert den Index.[20]

[20]Daher findet man oft die Definition: ein Array $A[1..n]$ heißt Minimum-Heap, wenn für alle $i \geq 1$ gilt,

Wir werden den Heap in demselben Array speichern, in dem die zu sortierenden Elemente bereits stehen – es wird so kein zusätzlicher Speicher oder ein Kopieren der Elemente erforderlich sein.

Für die Implementierung muss man sich nun nur noch überlegen, wohin man die dem Heap entnommenen Elemente hinspeichert. Hierbei bedient man sich eines einfachen Tricks: Der Heap wird in der Sortierphase mit jedem Entnehmen eines Elementes kleiner – im Array gehören so die Elemente am Ende des Arrays nach und nach nicht mehr zum Heap. Dort sollen nach dem Sortieren aber die größten Elemente stehen. Die Lösung: man benutzt einen Maximum-Heap, bei dem jeweils die größten Elemente in der Wurzel stehen. So kann ein Element bei Entnahme einfach an die im Heap freiwerdende Stelle getauscht werden.

Machen Sie sich das Prinzip an einem Beispiel mit wenigstens 7 Zahlen klar und überprüfen Sie die Korrektheit anhand des folgenden Programmcodes (lassen Sie sich z. B. nach jedem sink(.,.)-Aufruf (Zeilen 35 und 39) den Inhalt des Arrays ausgeben). Übung

`sortieren.cpp` (Fortsetzung)

```
int *heap; // modulweit bekanntes Array für Heapsort

void sink(const int i, const int n) { // lässt das Element heap[i]
  // im heap[1..n] so absinken, dass die Heapeigenschaft erfüllt ist
  int v=i; int l,r; int h1,h2;
  // bestimme Blatt des Einsinkpfades
  while (2*v<n) {
    l=2*v; r=l+1;
    if (heap[l] > heap[r]) { // sinkt in Richtung des größeren Kindes
      v=l;
    } else {
      v=r;
    }
  }
  if (2*v==n) {
    v=n; // Einzelkind
  }
  // v ist jetzt das Blatt des Einsinkpfades
  // im Einsinkpfad soweit nötig wieder aufsteigen
  while (heap[v] < heap[i]) { v=v/2; }
  // Elemente auf dem Pfad bis i rotieren
  h1=heap[v]; heap[v]=heap[i]; v=v/2;
  while (v > i) {
    h2=h1; h1=heap[v]; heap[v]=h2; v=v/2;
  }
  heap[i]=h1;
}

void HeapSort(int tosort[],const int l,const int r) { //sortiert tosort[l..r]
  heap=tosort+l-1; // heap zeigt jetzt auf das Element links vom linken Rand
  // auf heap[0] darf nicht zugegriffen werden!
  int n=r-l+1; // soviel Elemente werden sortiert
  // der zu sortierende Bereich steht jetzt in heap[1..n]
  // Heapaufbau
  for(int i=n/2;i>=1;i--) { sink(i,n); }
```

dass $A[i] \leq A[2i]$, $2i \leq n$ und $A[i] \leq A[2i+1]$, $2i+1 \leq n$. Oder anders formuliert: ein Array $A[1..n]$ heißt Minimum-Heap, wenn für alle $i \in \{2,\ldots,n\}$ gilt, dass $A[i/2] \leq A[i]$. Analog mit „$\geq$“ und Maximum-Heap.

```
    // Sortierphase
    for(int i=n;i>1;i--) {
      tausche(heap[i],heap[1]);
      sink(1,i-1);
    }
}
```

Ein einzelnes Absinkenlassen eines Elementes benötigt höchstens 2·(Höhe des Unterbaums −1) Vergleiche. Beim Heapaufbau müssen $n/4$ Unterbäume der Höhe 2, $n/8$ Unterbäume der Höhe 3, ... und 1 Unterbaum der Höhe $\log n$ betrachtet werden. Der Aufwand beträgt also

$$\sum_{i=2}^{\log n} \frac{n}{2^i} \cdot 2(i-1) = n \cdot \sum_{i=2}^{\log n} \frac{i-1}{2^{i-1}} < n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = n \cdot \sum_{i=1}^{\infty} \sum_{j=i}^{\infty} \frac{1}{2^j} = n \cdot \sum_{i=1}^{\infty} \frac{2}{2^i} = 2n$$

In der Sortierphase werden pro Element höchstens $2 \log n$ Vergleiche benötigt (unter Berücksichtigung der kleiner werdenden Heaps erhält man in der Summe $2n \log n - 4n$ Vergleiche).

Dieser Algorithmus, das Bottom-Up-Heapsort, ist sehr effizient. Es benötigt nur konstant viel zusätzlichen Speicherplatz und kommt im Mittel mit nur $n \log n + 0.4n$ Vergleichen aus, wobei die Schwankungen um dieses Mittel sehr gering sind. Eine grobe Abschätzung des Worst-Cases führt wie oben gezeigt auf $2n \log n$ Vergleiche. Bei genauerer Untersuchung – wir verweisen hier auf die Literatur – lässt sich auch eine Grenze von $1.5n \log n$ herleiten (dies liegt daran, dass nicht alle Elemente im Heap wieder weit vom Blatt des Einsinkpfades aufsteigen können).

Bei komplizierten Datenstrukturen, für die die Vergleiche gegenüber den Indexberechnungen teuer sind, ist Bottom-Up-Heapsort dem meist verwendeten Quicksort überlegen. Sind jedoch nur einfache Ganzzahlen zu sortieren, ist Quicksort im Mittel etwas besser.

## 2.4 Zusammenfassung

Abhängig vom Anwendungsgebiet sollte man das Sortierverfahren konkret auswählen:

- In stark vorsortierten Feldern, bei denen also der Fehlstand gering ist, kann man Insertionsort verwenden (wichtig: in der hier vorgestellten Variante! – siehe Anmerkung oben im Text).
- Spielt der Speicherverbrauch keine Rolle, ist Mergesort das effizienteste Verfahren.
- Spielt Speicherverbrauch eine Rolle (und liegen komplizierte zu sortierende Datenstrukturen vor), ist Bottom-Up-Heapsort zu empfehlen.
- Zum Sortieren einfacher Datenstrukturen bei nicht vorsortierten Feldern ist Quicksort im Mittel am schnellsten.

Führen Sie selbst Experimente durch, die für verschiedene Ausgangssituationen die Laufzeiten der Verfahren miteinander vergleichen. Vergleichen Sie diese insbesondere auch mit dem Sortier-Algorithmus aus der Standard-Bibliothek – können Sie aus den ermittelten Laufzeiten erahnen, welches Verfahren in der Standard-Bibliothek verwendet wird? Übung

# 3 Graphalgorithmen

. . . noch in Arbeit . . .

Der C/C++-Code ist umfangreich dokumentiert, vollziehen Sie die Themen am Code nach.

## 3.1 Datenstrukturen

- Adjazenzmatrix
- Adjazenzliste

## 3.2 Tiefen- und Breitensuche

- Tiefensuche – Stack Reloaded
- Breitensuche – Queue Reloaded

## 3.3 Berechnung kürzester Wege

- Dijkstra-Algorithmus
  - Randverwaltung als Liste
  - Randverwaltung als Heap

graphen.cpp

```
// sl-4-BA-Inf2: Datenstrukturen für Graphen (Adjazenzmatrix, Adjazenzliste)
// Graphalgorithmen: Tiefensuche, Breitensuche, kürzeste Wege (Dijkstra)

// Begriffe:
// gerichteter Graph G=(V,E) mit Knotenmenge V und Kantenmenge E aus VxV
// (ungerichteter Graph G=(V,E), Richtung der Kanten weglassen)
// Kante (x,y) ist inzident zu ihren Knoten x und y; x und y sind adjazent/benachbart
// S(x) Menge der Nachfolger = { y | (x,y) in E }
// P(x) Menge der Vorgänger = { y | (y,x) in E }
// N(x) Menge der Nachbarn = { y | (x,y) in E oder (y,x) in E }
// d+(x) = |S(x)| Ausgangsgrad, d-(x) = |P(x)| Eingangsgrad, d(x) = d+(x) + d-(x) Grad
// (x,x) heißt Schlinge
// Adjazenzmatrix A=(a(i,j)) zum Graphen G=(V,E) mit V={v1,...,vn},
// a(i,j)=1 gdw. (vi,vj) ist Kante in E
// gewichtete Graphen G=(V,E,c), c:E->R
// in diesem Programm Adjazenzmatrix eines gewichteten Graphen
// a(i,j)=x, x!=0 gdw. (vi,vj) ist Kante in E mit Gewicht c(vi,vj)=x
// Adjazenzliste: Liste von Knoten, jeder Knoten speichert Liste seiner Nachfolger

// In der Regel liegen Graphen als Adjazenzlisten vor, in diesem Programm wird
// Darstellung als Matrix hauptsächlich zur einfachen Generierung der
// Zufallsgraphen verwendet sowie als weitere Datenstruktur zur möglichen
// Darstellung von Graphen im Rechner

// Übungsaufg.: Wandeln Sie (von Hand) Adjazenzmatrizen in Adjazenzlisten um,
```

```
// "malen" Sie die Graphen und führen Sie eine Tiefen- und Breitensuche durch.
// Bestimmen Sie (von Hand) mit Dijkstras Algorithmus die kürzesten Wege
// zwischen allen Paaren von Knoten (wie oft muss dazu der Dijkstra-Algorithmus
// ausgeführt werden?)

#include "time.h"
#include <iostream>
using namespace std;

const int ANZ_KNOTEN = 7;

int ZufaZa() {
  return rand()%9 + 1; // erzeugt eine 1-stellige Zufallszahl != 0
  // return rand()%90 + 10; // erzeugt eine 2-stellige Zufallszahl
}

int AdjMat[ANZ_KNOTEN][ANZ_KNOTEN];

void InitAdjMat(int mat[ANZ_KNOTEN][ANZ_KNOTEN]) {
  // sehr(!!!) einfacher Zufallsgraphgenerator
  for(int i=0; i<ANZ_KNOTEN; i++)
    for(int j=0; j<ANZ_KNOTEN; j++)
      mat[i][j] = (i!=j && rand()%100<35) ? ZufaZa() : 0 ;
  // keine Schlingen und restliche Kanten mit Wahrscheinlichkeit 35%
}

void CoutAdjMat(int mat[ANZ_KNOTEN][ANZ_KNOTEN]) {
  cout << "Adjazenzmatrix:" << endl;
  for(int i=0; i<ANZ_KNOTEN; i++) {
    for(int j=0; j<ANZ_KNOTEN; j++)
      cout << AdjMat[i][j] << ' ';
    cout << endl;
  }
}

struct Edge;

struct Node { // hier: id = Index im Array of Nodes
  int dfsnr; // Nummer, die der Knoten bei der Tiefensuche erhält
  int bfsnr; // Nummer, die der Knoten bei der Breitensuche erhält
  Edge* EIK; // Erste inzidente Kante = Beginn seiner Nachfolgerknotenverweisliste
  void init(int idfsnr, int ibfsnr, Edge* iEIK) {
    dfsnr=idfsnr; bfsnr=ibfsnr; EIK=iEIK;
  }
};

struct Edge {
  int cost; // Gewicht der Kante
  int EKn; // Index des Endknotens der Kante
  Edge* NKa; // nächste inzidente Kante
  void init(int icost, int iEKn, Edge* iNKa) {
    cost=icost; EKn=iEKn; NKa=iNKa;
  }
};
```

```
Node AdjLst[ANZ_KNOTEN]; // könnte etwas allgemeiner (und komplizierter)
// auch als Liste mit Pointern implementiert werden

void AdjMat2Lst(int mat[ANZ_KNOTEN][ANZ_KNOTEN], Node lst[ANZ_KNOTEN]){
  // jede Zeile der Adjazenzmatrix durchgehen und die
  // Einträge != 0 als Kanten einfügen
  for(int i=0;i<ANZ_KNOTEN;i++) {
    for(int j=ANZ_KNOTEN-1;j>=0;j--) { // Kanten werden jeweils vorne eingefügt,
      // so ist die Reihenfolge in der Liste gleich der in der Matrix
      if (mat[i][j]) {
  Edge* newedge = new Edge; newedge->init(mat[i][j],j,lst[i].EIK);
  lst[i].EIK=newedge;
      }
    }
  }
}

void CoutAdjLst(Node lst[ANZ_KNOTEN]) {
  cout << "Adjazenzliste:" << endl;
  for(int i=0;i<ANZ_KNOTEN;i++) {
    Edge* ptr = lst[i].EIK;
    while (ptr) {
      cout << "Kante von " << i << " nach " << ptr->EKn
           << " mit Gewicht " << ptr->cost << endl;
      ptr=ptr->NKa;
    }
  }
}

int dfscount=1; // globaler Zähler für die Tiefensuche
// muss bei weiterer Tiefensuche neu initialisiert werden
// dito die "dfsnr" in der "AdjLst"

void Tiefensuche(int i) { // Schema: falls Knoten i noch nicht markiert ist,
  // markiere Knoten i und führe die Tiefensuche von allen Nachfolgern aus aus.
  if (!AdjLst[i].dfsnr) { // noch nicht markierte Knoten haben noch dfsnr=0
    AdjLst[i].dfsnr=dfscount++;
    Edge* ptr = AdjLst[i].EIK;
    while (ptr) {
      Tiefensuche(ptr->EKn);
      ptr=ptr->NKa;
    }
  }
}

void CoutDFSNumbers(Node lst[ANZ_KNOTEN]) {
  cout << "DFS-Nummern:" << endl;
  for(int i=0;i<ANZ_KNOTEN;i++) {
    cout << "Knoten " << i << " hat DFS-Nr. " << lst[i].dfsnr << endl;
  }
}

// für die Breitensuche werden Queues benötigt (vom Anfang des Semesters kopiert)

struct queue { // als Ringliste: zeigt auf letztes Element, queue->front auf erstes
```

```
  int inhalt;
  queue *front;
};

void enqueue (queue *& q, int value) {
  if(q==0) {
    q = new queue; // erstes Element in der Queue
    q->inhalt = value;
    q->front = q; // erstes und letztes Element sind identisch
  } else {
    queue * elem = new queue;
    elem->front = q->front;
    elem->inhalt = value;
    q->front = elem;
    q = elem;
  }
}

int dequeue (queue *& q) { // geht davon aus, dass die Queue q nicht leer ist
  // die aufrufende Funktion muss ggf. über is_empty(q) (also q==0)
  // testen, ob die Queue q leer ist
  int value = q->front->inhalt;
  if (q->front==q) { // ist es das letzte Element in der Queue?
    delete q;
    q=0;
  } else {
    queue *old = q->front;
    q->front=q->front->front;
    delete old;
  }
  return value;
}

bool is_empty (queue * q) {
  return q==0;
}

void Breitensuche(int i) { // Schema: beginne mit Knoten i
  // markiere Knoten i und merke in der Queue alle Nachfolger
  // entnehme jeweils den nächsten Knoten der Queue und merke,
  // falls dieser noch nicht markiert ist, alle Nachfolger in der Queue
  int bfscount=1; // Zähler für die Breitensuche
  int idx;
  queue* q=0;
  enqueue(q,i);
  while (!is_empty(q)) {
    idx=dequeue(q);
    if (!AdjLst[idx].bfsnr) { // noch nicht markierte Knoten haben noch bfsnr=0
      AdjLst[idx].bfsnr=bfscount++;
      Edge* ptr = AdjLst[idx].EIK;
      while (ptr) {
        enqueue(q,ptr->EKn);
        ptr=ptr->NKa;
      }
    }
```

```
  }
}

void CoutBFSNumbers(Node lst[ANZKNOTEN]) {
  cout << "BFS-Nummern:" << endl;
  for(int i=0;i<ANZKNOTEN;i++) {
    cout << "Knoten " << i << " hat BFS-Nr. " << lst[i].bfsnr << endl;
  }
}

// Datenstrukturen für Dijkstras Algorithmus

// Idee von Dijkstras Algorithmus (kürzeste Wege in Graphen mit Kantengewichten >=0):
// Die Knoten werden intern aufgeteilt in Baumknoten (Knoten, für die
// die kürzeste Entfernung garantiert gefunden wurde),
// Randknoten (Knoten, die nicht Baumknoten aber Nachfolger eines Baumknotens
// sind -- für diese ist ein Weg (nicht notwendigerweise der kürzeste) gefunden),
// sowie der Rest. Im Programm werden explizit nur die Randknoten verwaltet
// (die Baumknoten sind die Knoten, die nicht im Rand sind und für die schon ein Weg
// gefunden wurde; der Rest besteht aus den Knoten, die noch nicht erreicht wurden)
// Für jeden Knoten im Rand wird sich der bisher kürzeste gefundene Weg
// (genauer: dessen Entfernung) gemerkt.
// Ablauf: Zu Beginn ist der Startknoten in B und dessen Nachfolger in R, gefundene
// Entfernungen sind 0 zum Startknoten und das jeweilige Kantengewicht zu den Nachfolgern
// In jeder Iteration wähle den Knoten aus R, dessen gefundene Entfernung minimal ist
// und prüfe, ob über diesen Knoten dessen Nachfolger auf kürzerem Weg erreicht werden.
// Korrektheit: Vollständige Induktion über die Anzahl der Iterationen:
// Zu Beginn ist der kürzeste Weg zum Startknoten (Länge 0) gefunden.
// Bei jeder Auswahl des Knotens mit minimaler Entfernung aus R ist diese die
// garantiert kürzeste, weil über andere Knoten (Entfernung zu diesen ist mindestens
// genauso groß) und weitere Kanten (Gewichte >=0) der Weg höchstens länger werden kann.

// Um nicht nur die kürzesten Entfernungen zu bestimmen, sondern auch den zugehörigen
// Weg angeben zu können, kann man einen Kürzeste-Wege-Baum berechnen lassen.
// Dazu speichert man sich jedes Mal, wenn ein Knoten in den Baum eingefügt wurde
// oder sich die gefundene Entfernung verringert hat, den Knoten, über den dieser
// kürzere Weg gefunden wurde  (= der zuletzt über delete_min aus R entnommene Knoten)
// (-> Array int pred[ANZKNOTEN]) -- Beispiel: Führt der kürzeste Weg von
// 0 nach 3 über die Knoten 6, 2 und 5, so wurde beim für den Knoten 6 sich der
// Vorgänger 0 gemerkt, für den Knoten 2 der Vorgänger 6, für Knoten 5 der Vorgänger 2
// und für den Knoten 3 der Vorgänger 5. Die Vorgänger können nun ausgehend vom Knoten 3
// bestimmt werden (0 hat keinen Vorgänger, z.\,B.==-1) (also 3,5,2,6,0) und dann rückwärts
// ausgegeben werden -- Aufgabe: Implementieren Sie die Berechnung und Ausgabe der Wege

// 1. Variante: Verwaltung der Randmenge R als Liste (struct RLst)
// delete_min (Suche des Knotens mit minimaler Entfernung in R):
// -> Liste durchlaufen: Aufwand O(n)
// insert (Einfügen eines neuen Knotens in R):
// -> Vorne in der Liste einfügen: Aufwand O(1)
// decrease_key (Verringern der Entfernung eines Knotens, der schon in R ist):
// -> direkter Zugriff auf das Element in der Liste: Aufwand O(1)
// Gesamtaufwand: n*delete_min + m*(insert oder decrease_key) = O(n^2)

struct RLst_Node {
  int node; // Index im Knotenarray
```

```
  int d; // Entfernung zum Startknoten
  RLst_Node* next; // nächstes Element in der Liste
};

struct RLst {
  RLst_Node* whereinR[ANZ_KNOTEN]; // Verweis, wo der Knoten in der Liste sich befindet
  RLst_Node* lst; // erstes Element in der Liste
  void init() { lst=0; for(int i=0;i<ANZ_KNOTEN;i++) whereinR[i]=0; }
  bool is_empty() { return (lst==0); }
  bool member(int node) { return (whereinR[node]!=0); }
  void insert(int node, int d) { // fügt Knoten i mit Wert d am Anfang von lst ein
    RLst_Node *elem = new RLst_Node;
    elem->node = node; elem->d = d; elem->next = lst;
    lst = elem; whereinR[node]=lst;
  }
  int delete_min() { // gibt Index des Knotens mit kleinstem Wert d in lst zurück
    // kleinstes Element finden
    RLst_Node *min = lst; // Zeiger auf Listenelement mit bisher kleinstem Wert
    RLst_Node *ptr = lst->next; // Pointer zum Durchlaufen der Liste
    while (ptr) {
      if (ptr->d < min->d) min=ptr; // Minimum merken
      ptr=ptr->next;
    }
    int rueckgabewert = min->node;
    // Element löschen
    if (min==lst) { // erstes und/oder einziges Element
      lst=lst->next;
      delete min;
      whereinR[rueckgabewert]=0;
    } else { // Vorgänger von min finden und min rauslöschen
      ptr=lst;
      while (ptr->next!=min) ptr=ptr->next;
      ptr->next=min->next;
      delete min;
      whereinR[rueckgabewert]=0;
    }
    // Index zurückgeben
    return rueckgabewert;
  }
  void decrease_key(int node, int d) { // verringert den Wert von Knoten node auf d
    whereinR[node]->d=d;
  }
  void debug() { cout << "R="; RLst_Node* ptr=lst;
    while (ptr) { cout << " " << ptr->node << ":" << ptr->d; ptr=ptr->next; }
    cout << endl;
  }
};

// 2. Variante: Verwaltung der Randmenge R als Heap (struct RHeap)
// delete_min (Suche des Knotens mit minimaler Entfernung in R):
// -> in der Wurzel des Heaps, Heapeigenschaft wieder herstellen: Aufwand O(log(n))
// insert (Einfügen eines neuen Knotens in R):
// -> Am Ende vom Heap einfügen und aufsteigen lassen: Aufwand O(log(n))
// decrease_key (Verringern der Entfernung eines Knotens, der schon in R ist):
// -> Element im Heap aufsteigen lassen: Aufwand O(log(n))
```

```
// Gesamtaufwand: n*delete_min + m*(insert oder decrease_key) = O(m*log(n))
// In Straßengraphen ist m<=c*n -> Aufwand mit Heaps deutlich geringer als mit Listen

struct RHeap {
  int whereinR[ANZ_KNOTEN]; // Index, an dem sich der Knoten im Heap befindet
  int last; // letzter Index, der zum Heap gehört
  struct { int node; int d; } heap[ANZ_KNOTEN];
  // min-Heap!: Vater <= Soehne bzgl. der d-Werte
  void init() { last=-1; for(int i=0;i<ANZ_KNOTEN;i++) whereinR[i]=-1;
  } // heap wird beim Einfügen initialisiert
  bool is_empty() { return (last==-1); }
  bool member(int node) { return (whereinR[node]!=-1); }
  void insert(int node, int d) {//fügt Knoten i mit Wert d als neues Element im Heap ein
    // und lässt ihn an die richtige Stelle aufsteigen
    last++;
    int sohn=last; // Index des Sohnknotens
    int vater=(sohn-1)/2; // Index des Vaterknotens
    while (sohn>0 && heap[vater].d > d) { // sohn = Index der "freien Stelle"
      // wenn sohn==0, dann gibt es keinen Vater, der noch runterrutschen kann
      heap[sohn].node=heap[vater].node;
      heap[sohn].d   =heap[vater].d   ;
      whereinR[heap[vater].node]=sohn;
      sohn=vater; vater=(sohn-1)/2;
    }
    heap[sohn].node=node; heap[sohn].d=d; whereinR[node]=sohn;
  }
  int delete_min() { // gibt Index des Knotens mit kleinstem Wert d im Heap zurück
    // kleinstes Element ist in der Wurzel, dieses löschen und Element von
    // Position last auf die Wurzel kopieren und in Heap[0..--last] einsinken lassen
    int rueckgabewert = heap[0].node; whereinR[rueckgabewert]=-1;
    int node=heap[last].node; int d=heap[last].d; // (node,d) sinkt ein
    last--; // Heapgröße um 1 verkleinert
    int vater=0; int links=2*vater+1; int rechts=links+1;
    while (links<=last) { // es existiert noch mind. 1 Kind
      if (rechts<=last) { // beide Kinder existieren
  if (heap[links].d>heap[rechts].d) links=rechts;
  // heap[links].d ist jetzt der "kürzere" Sohn
  if (heap[links].d<d) { // Element sinkt weiter
    heap[vater].node=heap[links].node;
    heap[vater].d   =heap[links].d   ;
    whereinR[heap[vater].node]=vater;
    vater=links; links=2*vater+1; rechts=links+1;
  } else break; // Element sinkt nicht weiter
      } else { // links=last und das linke Kind ist ein Blatt
  if (heap[links].d<d) {
    heap[vater].node=heap[links].node;
    heap[vater].d   =heap[links].d   ;
    whereinR[heap[vater].node]=vater;
    vater=links;
  }
  break; // Element sinkt nicht weiter
      }
    }
    // heap[vater] ist jetzt der freie Platz
    heap[vater].node=node;
```

```
    heap[vater].d    =d    ;
    whereinR[heap[vater].node]=vater;
    // Index zurückgeben
    return rueckgabewert;
  }
  void decrease_key(int node, int d) { // verringert den Wert von Knoten node auf d
    heap[whereinR[node]].d=d;
    int sohn=whereinR[node]; // Index des Sohnknotens
    int vater=(sohn-1)/2; // Index des Vaterknotens
    while (sohn>0 && heap[vater].d > d) { // sohn = Index der "freien Stelle"
      // wenn sohn==0, dann gibt es keinen Vater, der noch runterrutschen kann
      heap[sohn].node=heap[vater].node;
      heap[sohn].d    =heap[vater].d    ;
      whereinR[heap[vater].node]=sohn;
      sohn=vater; vater=(sohn-1)/2;
    }
    heap[sohn].node=node; heap[sohn].d=d; whereinR[node]=sohn;
  }
  void debug() { cout << "R="; for(int i=0;i<=last;i++)
    cout << " " << heap[i].node << ":" << heap[i].d;
    cout << endl; }
};

void Dijkstra(Node lst[ANZ_KNOTEN], int s) { // berechnet die kürzesten Entfernungen
  // vom Knoten s zu allen anderen Knoten
  int d[ANZ_KNOTEN]; // Entfernungen von s zu .
  int u,v; // Knoten-Indizes
  int c; // Weglänge
  Edge* nachfolger;
  const int infty=ANZ_KNOTEN*10;//10>größtes Gewicht, maximal ANZ_KNOTEN-1 Kanten pro Weg
  // Initialisierung
  for (int i=0;i<ANZ_KNOTEN;i++) d[i]=infty; // = noch unbekannt = unendlich
  d[s]=0;
  // RLst R;  // EINE DER BEIDEN ZEILEN AUSKOMMENTIEREN
  RHeap R; // UM ZWISCHEN R ALS LISTE UND R ALS HEAP ZU WÄHLEN
  R.init();
  R.insert(s,0); // vereinfachte Initialisierung, im ersten Schritt
  // wird s aus R ausgewählt und die Nachfolger in R eingefügt
  // Berechnung
  while (!R.is_empty()) { // solange noch neue Knoten erreichbar sind
    u=R.delete_min(); // wähle den Knoten mit kleinster Entfernung zum Startknoten
    nachfolger=lst[u].EIK;
    while (nachfolger!=0) { // teste für alle ausgehenden Kanten
      v = nachfolger->EKn; c = d[u]+nachfolger->cost; // Weglänge nach v über u
      if (c<d[v]) {
  d[v]=c;
  if (!R.member(v)) R.insert(v,c);
  // falls der Nachfolger noch nie betrachtet wurde, füge ihn in R ein
  else R.decrease_key(v,c);
  // falls der Weg zum Nachfolger über u kürzer ist, verringere die Entfernung
  // zum Nachfolger auf den entsprechend kleineren Wert
      }
      nachfolger=nachfolger->NKa;
    }
  }
```

```
  // Ausgabe
  cout << "Errechnete Entfernungen:" << endl;
  for (int i=0;i<ANZ_KNOTEN;i++)
    if (d[i]==infty)
      cout << "Knoten " << i << " ist von Knoten "
           << s << " aus nicht erreichbar!" << endl;
    else
      cout << "Entfernung von Knoten " << s << " zum Knoten "
           << i << " betraegt " << d[i] << endl;
}


int main()
{
  srand((unsigned) time(NULL)); // Initialisierung des Zufallszahlengenerators

  InitAdjMat(AdjMat);
  CoutAdjMat(AdjMat);

  AdjMat2Lst(AdjMat,AdjLst);
  CoutAdjLst(AdjLst);

  Tiefensuche(0);
  CoutDFSNumbers(AdjLst);

  Breitensuche(0);
  CoutBFSNumbers(AdjLst);

  Dijkstra(AdjLst,0);

  system("Pause");
  return EXIT_SUCCESS;
}
```

# 4 Entwurfsstrategien – ein Überblick

Das Wesen des Problems führt zur Wahl der Entwurfsmethodik. Fassen wir die Entwurfsstrategien nochmals zusammen:

- Zeigt das Problem keine Strukturen, keine Optimalitätseigenschaften für Teilmengen oder ähnliches, so bleibt einem nur systematisch alle möglichen Elemente des Lösungsraumes zu untersuchen. In der Regel geht man dabei rekursiv vor und nimmt bei Erkennen von Sackgassen den letzten Schritt zurück und probiert Alternativen. Das Verfahren nennt man Backtracking. Das klassische Beispiel ist die gerechte Erbschaft. Verwandt ist das Verfahren Branch-and-Bound, ein Backtracking-Verfahren, bei dem man versucht, Teilberechnungen, die zu keiner Lösung mehr führen können, frühzeitig abzubrechen. Sowohl Backtracking als Branch-and-Bound-Algorithmen haben typischerweise exponentiellen Aufwand.

  *Backtracking, Branch-and-Bound*

- Bei der dynamischen Programmierung versucht man aus Lösungen für Teilprobleme

  *Dynamisches Programmieren*

die Lösung des Gesamtproblems zu bestimmen. Dabei stellt sich erst im Laufe des Algorithmus heraus, welche Lösungen der Teilprobleme tatsächlich für die Gesamtlösung gebraucht werden. Das klassische Beispiel ist die Bestimmung optimaler Suchbäume. Oft bestehen die Programme aus drei geschachtelten Schleifen (über die Größe der Teilprobleme, die Teilprobleme dieser Größe und eine Auswahl des Optimums für das Teilprobleme aus einer Menge von Kandidaten), der Aufwand beträgt oft $O(n^3)$.

- Während sich beim dynamischen Programmieren eine Lösung aus optimalen Teillösungen zusammensetzt (wir jedoch die zugehörigen „Teile“ nicht vorab kennen), lassen sich bei Greedy-Algorithmen Teillösungen schrittweise optimal erweitern. Es finden keinerlei unnötige Berechnungen statt. Das klassische Beispiel ist die Berechnung kürzester Entfernungen mit Dijkstras Algorithmus. Der Aufwand ist meist $O(n^2)$ oder besser. *Greedy-Algorithmen*

- Divide-and-Conquer-Verfahren lassen sich immer dann anwenden, wenn ein einfaches Teilen in unabhängige Teilprobleme möglich ist und sich diese Teillösungen leicht zur Gesamtlösung zusammensetzen lassen. Klassische Beispiele sind Mergesort und Quicksort. Realisiert werden diese Ansätze meist durch Rekursion. Ist der Aufwand zum Aufteilen in und Zusammenfügen der Teillösungen durch $O(n)$ beschränkt, so beträgt die Laufzeit (bei etwa gleichgroßen Teilproblemen) $O(n \log n)$. *Divide-and-Conquer*

# A Zusätzliche Algorithmen in C/C++

Hier sind noch die zur Verfügung gestellten C/C++-Programme angegeben, die in der Vorlesung nicht weiter besprochen werden.

## A.1 Visualisierung von Binärbäumen

Neben der Visualisierung einfacher binärer Suchbäume werden auch die im Abschnitt 1.3.8 vorgestellten AVL-Bäume visualisiert, indem zunächst eine Kopie des AVL-Baums erstellt wird. Um den Speicher dieser Kopie später wieder freigeben zu können, hat unser `suchbaum`-Modul noch eine Löschfunktion erhalten:

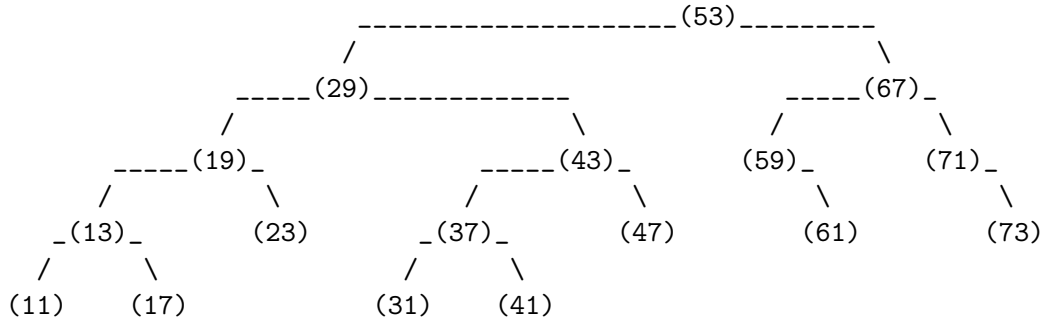
`suchbaum.cpp` (Fortsetzung)

```
void delTree(const bstPtr root) {
  if (root!=NULL) {
    delTree(root->left);
    delTree(root->right);
    delete root;
  }
}
```

Das eigentliche Modul zur Visualisierung zeigt nebenbei auch, wie man den cout-Operator erweitern kann, um eigene Datentypen wie z. B. Verbunde oder – wie hier – Bäume ausgeben zu können. Das Umschreiben in eine in C verwendbare Funktion ist eine kleine Fleissübung.

Die Idee der Visualisierung ist, zunächst über einen Inorderdurchlauf für jeden Knoten zu bestimmen, das wievielte Element der Knoten in der sortierten Folge ist (soweit wird er links vom Rand dargestellt – siehe Beispiel: je „Spalte“ wird genau ein Knoten dargestellt), danach

wird der Baum levelweise durchlaufen und jeweils aufgrund der Informationen des vorhergehenden Levels die Kanten des Baumes als Unter- und Schrägstriche dargestellt. Die Breite der Darstellung ist auf Zahlen zwischen 0 und 99 ausgelegt.




visbinbaum.h

```
#ifndef VISUALBINBAUM_H
#define VISUALBINBAUM_H

#include <cstdlib>
#include <iostream>
#include "suchbaum.h"
#include "avlbaum.h"

using namespace std;

ostream& operator << (ostream& os, const bstPtr root);
ostream& operator << (ostream& os, const avlPtr root);
// to be used for "cout << tree;" and "cerr << tree;"

#endif
```

```
#include <cstdlib>
#include <iostream>
#include "suchbaum.h"
#include "avlbaum.h"
#include "visualbinbaum.h"

using namespace std;

////////////////////////////////////////////////////////////////////////
struct extBsTree; // Binbaum mit Zusatzinfos
typedef extBsTree * extBstPtr;
struct extBsTree {
  DataType value;         // Kopie
  extBstPtr left, right; // "Kopie"
  int inordernr;          // Position in sortierter Rhf
  bool isleft;            // Flag, ob es ein linkes Kind ist
};
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
extBstPtr cpBsTree(const bstPtr root, const bool isleft, int & inordernr) {
////////////////////////////////////////////////////////////////////////
  // Kopiert den Baum und berechnet Abstand vom linken Rand
  extBstPtr cproot, cpleft;
  if (root==NULL) {
    return NULL;
  } else {
    cpleft=cpBsTree(root->left,true,inordernr);
    inordernr++;
    cproot= new extBsTree;
    cproot->value=root->value;
    cproot->left=cpleft;
    cproot->inordernr = inordernr;
    cproot->isleft=isleft;
    cproot->right=cpBsTree(root->right,false,inordernr);
    return cproot;
  }
}

////////////////////////////////////////////////////////////////////////
void delTree(const extBstPtr root) {
////////////////////////////////////////////////////////////////////////
  if (root!=NULL) {
    delTree(root->left);
    delTree(root->right);
    delete root;
  }
}
```

```
50 ////////////////////////////////////////////////////////////////////////////
   DataType MaxExtBsTree(const extBstPtr root) {
   ////////////////////////////////////////////////////////////////////////////
     // root mustn't be NULL
     if (root->right==NULL) {
55     return root->value;
     } else {
       return MaxExtBsTree(root->right);
     }
   }
60
   ////////////////////////////////////////////////////////////////////////////
   struct extBstList; // will be used as mixed list and ring list
   typedef extBstList * extBstLPtr;
   struct extBstList { // für Levelorderausgabe
65   extBstPtr node;
     extBstLPtr next;
   };
   ////////////////////////////////////////////////////////////////////////////

70 ////////////////////////////////////////////////////////////////////////////
   void removeFirst (extBstLPtr & l, extBstPtr & node) {
   ////////////////////////////////////////////////////////////////////////////
     // l mustn't be empty
     extBstLPtr tbd = l;
75   node = l->node;
     l = l->next;
     delete tbd;
   }

80 ////////////////////////////////////////////////////////////////////////////
   void getListOfSuccessors(const extBstLPtr oldlist, extBstLPtr & newlist) {
   ////////////////////////////////////////////////////////////////////////////
     // recursion! at the end newlist must point to the first new item in that list
     // if no new item was added, newlist must be used unchanged in the next call
85   if (oldlist!=NULL) {
       if (oldlist->node->left!=NULL) {
         newlist = new extBstList;
         newlist->node = oldlist->node->left;
         newlist->next = NULL;
90       if (oldlist->node->right!=NULL) {
           newlist->next = new extBstList;
           newlist->next->node = oldlist->node->right;
           newlist->next->next = NULL;
           getListOfSuccessors(oldlist->next, newlist->next->next);
95       } else {
           getListOfSuccessors(oldlist->next, newlist->next);
         }
       } else {
```

```
      if (oldlist->node->right!=NULL) {
        newlist = new extBstList;
        newlist->node = oldlist->node->right;
        newlist->next = NULL;
        getListOfSuccessors(oldlist->next,newlist->next);
      } else {
        getListOfSuccessors(oldlist->next,newlist);
      }
    }
  } // else { // no more elements in oldlist
}

////////////////////////////////////////////////////////////////////////////
void mput(ostream& os, char bst, int times) {
////////////////////////////////////////////////////////////////////////////
  for(int i=0;i<times;i++) os << bst;
}


////////////////////////////////////////////////////////////////////////////
ostream& operator << (ostream& os, const bstPtr root) {
////////////////////////////////////////////////////////////////////////////
  // to be used for "cout << tree;" or "cerr << tree;"
  if (root==NULL) { os << "tree is empty!" << endl; return os; }

  extBstLPtr thisLevelList = NULL; // root of list
  extBstLPtr nextLevelList = NULL;
  extBstLPtr nextLevelNode = NULL; // current ptr to list
  extBstPtr thisNode = NULL;
  extBstPtr nextNode = NULL;

  int lastOffset = 0;
  int nrOfElements = 0;

  os << endl;

  // initialize List with (copied) root as the only top level node
  thisLevelList = new extBstList;
  thisLevelList->node = cpBsTree(root,false,nrOfElements);
  extBstPtr tbd=thisLevelList->node; // zur Speicherfreigabe am Ende
  thisLevelList->next = NULL;
```

```
  while (thisLevelList!=NULL) { // es gibt noch ein Level zum Ausgeben
    // nächstes Level bestimmen
    getListOfSuccessors(thisLevelList,nextLevelList);
    // unter Berücksichtigung des nächsten Levels das aktuelle ausgeben
    lastOffset=0; nextLevelNode=nextLevelList;
    while (thisLevelList!=NULL) {
      removeFirst(thisLevelList,thisNode);
      if (thisNode->left!=NULL) {
        nextNode=nextLevelNode->node; nextLevelNode=nextLevelNode->next;
        mput(os,' ',4*(nextNode->inordernr-lastOffset-1)+3);
        mput(os,'_',1+4*(thisNode->inordernr-nextNode->inordernr-1));
      } else {
        mput(os,' ',4*(thisNode->inordernr-lastOffset-1));
      }
      os << '('; if (thisNode->value<10 && thisNode->value>=0) { os << ' '; }
      os << thisNode->value << ')';
      lastOffset=thisNode->inordernr;
      if (thisNode->right!=NULL) {
        nextNode=nextLevelNode->node; nextLevelNode=nextLevelNode->next;
        mput(os,'_',4*(nextNode->inordernr-lastOffset-1)+1);
        mput(os,' ',3); lastOffset=nextNode->inordernr;
      }
    }
    os << endl;
    lastOffset=0; nextLevelNode=nextLevelList;
    while (nextLevelNode) {
      nextNode=nextLevelNode->node; nextLevelNode=nextLevelNode->next;
      if (nextNode) {
        mput(os,' ',4*(nextNode->inordernr-lastOffset-1));
        if (nextNode->isleft) os << "  /"; else os << " \\  ";
      }
      lastOffset=nextNode->inordernr;
    }
    os << endl;
    thisLevelList=nextLevelList; nextLevelList=NULL;
  }
  delTree(tbd); // Speicher der erweiterten Baumkopie freigeben
  return os;
}

////////////////////////////////////////////////////////////////////////////
ostream& operator << (ostream& os, const avlPtr root) {
////////////////////////////////////////////////////////////////////////////
  // to be used for "cout << tree;" or "cerr << tree;"
  bstPtr cp=avl2bst(root);
  os << cp;
  delTree(cp);
  cp=avlbal2bst(root);
  os << cp;
  delTree(cp);
  return os;
}
```

## A.2 Suchen, Einfügen und Löschen in AVL-Bäumen

Die Rotation haben wir bereits behandelt. Der grundsätzliche Aufbau der Funktionen ist identisch dem bei einfachen binären Suchbäumen. Man muss hier jedoch beim Rückwärtslaufen in der Rekursion jeweils die Balancen neu anpassen und ggf. (Doppel-)Rotationen durchführen. Dies macht den Code durch die notwendigen Fallunterscheidungen etwas länglich. Zudem benutzt der Code die Visualisierungsroutinen aus 1.3.5 / Anhang A.1 und beschreibt in jedem Schritt genau, was in welchem Unterbaum angepasst werden muss. In der Praxis wird man natürlich die ganzen Ausgaben weglassen (es geht dann ja um Datenverwaltung). Die Routinen sind dann wieder ausreichend kurz – ohne Kommentare im Modul wird man jedoch trotzdem schnell den Überblick verlieren.

*Fibonacci-Baum*

Als Zusatzfunktion ist noch eine Routine im Modul enthalten, die einen AVL-Baum mit möglichst großer Höhe bei möglichst wenigen Knoten erzeugt – dieser entspricht den Bäumen, die bei der Abschätzung der Höhe von AVL-Bäumen skizziert wurden. Aufgrund der Ähnlichkeit des Verhaltens zu dem der Fibonacci-Zahlen, werden diese speziellen Bäume meist Fibonacci-Bäume genannt.

avlbaum.h

```
#ifndef AVLBAUM_H
#define AVLBAUM_H

#include <cstdlib>
#include <iostream>
#include "suchbaum.h"

using namespace std;

enum avlBalance {unbalancedleft, balanced, unbalancedright};
struct avlTree;
typedef avlTree * avlPtr;
struct avlTree {
  DataType value;
  avlPtr left, right;
  avlBalance balance;
};

void avlInsert(avlPtr & root, const DataType value);
bool avlFind(const avlPtr root, const DataType value);
bool avlDelete(avlPtr & root, const DataType value);

void delTree(const avlPtr root);
bstPtr avl2bst(const avlPtr root);
bstPtr avlbal2bst(const avlPtr root);

void avlInsertFiboOfHeight(avlPtr & root, const int height);

#endif
```

```
#include <cstdlib>
#include <iostream>
#include "suchbaum.h"
#include "avlbaum.h"
#include "visualbinbaum.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////
void avl3PtrSwap(avlPtr &p1, avlPtr &p2, avlPtr &p3) {
////////////////////////////////////////////////////////////////////////////
  avlPtr ph=p1; p1=p2; p2=p3; p3=ph;
}

////////////////////////////////////////////////////////////////////////////
void avlRotate(avlPtr &root) {
////////////////////////////////////////////////////////////////////////////
  // balance changed by another one, so ...
  if (root->balance==unbalancedright) { // height difference is two now
    if (root->right->balance==unbalancedright) {
      cout << "Left rotation @ " << root->value
           << "--" << root->right->value << endl;
      avl3PtrSwap(root,root->right,root->right->left);
      root->balance=balanced; root->left->balance=balanced;
      cout << "Nodes " << root->left->value << " and "
           << root->value << " are now balanced" << endl;
    } else if (root->right->balance==unbalancedleft) {
      cout << "Right+left rotation" << endl;
      cout << "Right rotation @ " << root->right->left->value
           << "--" << root->right->value << endl;
      avl3PtrSwap(root->right,root->right->left,root->right->left->right);
      cout << "Subtree after right rotation "
           << "(balances will be corrected after the 2nd rotation):" << root;
      cout << "Left rotation @ " << root->value
           << "--" << root->right->value << endl;
      avl3PtrSwap(root,root->right,root->right->left);
      // new balanced depend on the former balance of the new root
      if (root->balance==unbalancedleft) {
        root->balance=balanced;
        root->left->balance=balanced;
        root->right->balance=unbalancedright;
        cout << "Nodes " << root->left->value << " and "
             << root->value << " are now balanced, node"
             << root->right->value << " is unbalanced to the right" << endl;
      } else if (root->balance==balanced) {
        root->balance=balanced;
        root->left->balance=balanced;
        root->right->balance=balanced;
        cout << "Nodes " << root->left->value << ", " << root->value
             << ", and " << root->right->value << " are now balanced" << endl;
```

```
      } else { // root->balance==unbalancedright
        root->balance=balanced;
        root->left->balance=unbalancedleft;
        root->right->balance=balanced;
        cout << "Node " << root->left->value
             << " is now unbalanced to the right, "
             << root->value << " and " << root->right->value
             << " are now balanced" << endl;
      }
    } else { // root->right->balance==balanced - can only happen during delete
      cout << "Left rotation @ " << root->value
           << "--" << root->right->value << endl;
      avl3PtrSwap(root,root->right,root->right->left);
      root->balance=unbalancedleft; root->left->balance=unbalancedright;
      cout << "Node " << root->left->value
           << " is now unbalanced to the right, and "
           << root->value << " is now unbalanced to the left" << endl;
    } // cases left and right+left rotation done
  } else { // root->balance==unbalancedleft // height difference is now -2
    if (root->left->balance==unbalancedleft) {
      cout << "Right rotation @ " << root->left->value
           << "--" << root->value << endl;
      avl3PtrSwap(root,root->left,root->left->right);
      root->balance=balanced; root->right->balance=balanced;
      cout << "Nodes " << root->value << " and "
           << root->right->value << " are now balanced" << endl;
    } else if (root->left->balance==unbalancedright) {
      cout << "Left+right rotation" << endl;
      cout << "Left rotation @ " << root->left->value
           << "--" << root->left->right->value << endl;
      avl3PtrSwap(root->left,root->left->right,root->left->right->left);
      cout << "Subtree after left rotation "
           << "(balances will be corrected after the 2nd rotation):" << root;
      cout << "Right rotation @ " << root->left->value
           << "--" << root->value << endl;
      avl3PtrSwap(root,root->left,root->left->right);
      // new balanced depend on the former balance of the new root
      if (root->balance==unbalancedleft) {
        root->balance=balanced;
        root->left->balance=balanced;
        root->right->balance=unbalancedright;
        cout << "Nodes " << root->left->value << " and "
             << root->value << " are now balanced, node"
             << root->right->value << " is unbalanced to the right" << endl;
      } else if (root->balance==balanced) {
        root->balance=balanced;
        root->left->balance=balanced;
        root->right->balance=balanced;
        cout << "Nodes " << root->left->value << ", " << root->value
             << ", and " << root->right->value << " are now balanced" << endl;
```

```
      } else { // root->balance==unbalancedright
        root->balance=balanced;
        root->left->balance=unbalancedleft;
        root->right->balance=balanced;
        cout << "Node " << root->left->value
             << " is now unbalanced to the left, "
             << root->value << " and " << root->right->value
             << " are now balanced" << endl;
      }
    } else { // root->right->balance==balanced - can only happen during delete
      cout << "Right rotation @ " << root->left->value
           << "--" << root->value << endl;
      avl3PtrSwap(root,root->left,root->left->right);
      root->balance=unbalancedright; root->right->balance=unbalancedleft;
      cout << "Node " << root->value << " is now unbalanced to the right, and "
           << root->right->value << " is now unbalanced to the left" << endl;
    } // cases right and left+right rotation done
  }
}

////////////////////////////////////////////////////////////////////////////
void avlInsertH(avlPtr & root, const DataType value, bool & higher) {
////////////////////////////////////////////////////////////////////////////
  if (root==NULL) {
    root = new avlTree;
    root->value = value;
    root->left = root->right = 0;
    root->balance = balanced;
    higher=true;
  } else if (value<root->value) {
    avlInsertH(root->left,value,higher);
    if (higher) {
      if (root->balance==unbalancedleft) {
        cout << "Rotation necessary @ " << root->value << " -- subtree:"<<root;
        avlRotate(root); higher=false;
        cout << "Tree is now avl again" << endl;
      } else if (root->balance==balanced) {
        root->balance=unbalancedleft; // higher is still true
        cout << "Node " << root->value
             << " is now unbalanced to the left" << endl;
      } else { // root->balance==unbalancedright
        root->balance=balanced; higher=false;
        cout << "Node " << root->value << " is now balanced" << endl;
      }
    }
  } else if (value>root->value) {
    avlInsertH(root->right,value,higher);
    if (higher) {
      if (root->balance==unbalancedleft) {
        root->balance=balanced; higher=false;
        cout << "Node " << root->value << " is now balanced" << endl;
```

```
      } else if (root->balance==balanced) {
        root->balance=unbalancedright; // higher is still true
        cout << "Node " << root->value
             << " is now unbalanced to the right" << endl;
      } else { // root->balance==unbalancedright
        cout << "Rotation necessary @ " << root->value << " -- subtree:"<<root;
        avlRotate(root); higher=false;
        cout << "Tree is now avl again" << endl;
      }
    }
  } else { // it is ==
    cout << "Element " << value << " is already in the tree!" << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
void avlInsert(avlPtr & root, const DataType value) {
////////////////////////////////////////////////////////////////////////////
  bool higher = false;
  avlInsertH(root,value,higher);
}

////////////////////////////////////////////////////////////////////////////
bool avlFind(const avlPtr root, const DataType value) {
////////////////////////////////////////////////////////////////////////////
  if (root==NULL) {
    return false;
  } else if (value<root->value) {
    return avlFind(root->left,value);
  } else if (value>root->value) {
    return avlFind(root->right,value);
  } else { // it is ==
    return true;
  }
}

////////////////////////////////////////////////////////////////////////////
void avlDeleteH(avlPtr &root,const DataType value,bool &lower,bool &deleted) {
////////////////////////////////////////////////////////////////////////////
  if (root==NULL) {
    cout << "Element " << value << " isn't in the tree!" << endl;
    lower=false;
    deleted=false; // no, the element was not in the tree
  } else if (value<root->value) {
    avlDeleteH(root->left,value,lower,deleted);
    if (lower) {
      if (root->balance==unbalancedleft) {
        root->balance=balanced;
        cout << "Node " << root->value << " is now balanced" << endl;
        // lower is still true
```

```
      } else if (root->balance==balanced) {
        root->balance=unbalancedright;
        lower=false;
        cout << "Node " << root->value
             << " is now unbalanced to the right" << endl;
      } else { // root->balance==unbalancedright
        if (root->right->balance==balanced) {
          lower=false;
        } // else { // lower is still true after rotation
        cout << "Rotation necessary @ " << root->value << " --- subtree:"<<root;
        avlRotate(root);
      }
    }
  } else if (value>root->value) {
    avlDeleteH(root->right,value,lower,deleted);
    if (lower) {
      if (root->balance==unbalancedleft) {
        if (root->left->balance==balanced) {
          lower=false;
        } // else { // lower is still true after rotation
        cout << "Rotation necessary @ " << root->value << " --- subtree:"<<root;
        avlRotate(root);
      } else if (root->balance==balanced) {
        root->balance=unbalancedleft;
        lower=false;
        cout << "Node " << root->value
             << " is now unbalanced to the left" << endl;
      } else { // root->balance==unbalancedright
        root->balance=balanced;
        cout << "Node " << root->value << " is now balanced" << endl;
        // lower is still true
      }
    }
  } else { // it is ==
    cout << "Element " << value << " is in the tree, I'll delete it!" << endl;
    if (root->left==NULL && root->right==NULL) {
      // it's a leaf => delete it
      delete root;
      root=NULL;
      lower=true; // balances will change on path to root
    } else if (root->left==NULL) {
      // right subtree must be non-empty, delete root and shift subtree up
      avlPtr tbd=root;
      root=root->right; // subtree has just one node, and it's balanced
      delete tbd;
      lower=true; // balances will change on path to root
    } else if (root->right==NULL) {
      // left subtree must be non-empty, delete root and shift subtree up
      avlPtr tbd=root;
      root=root->left; // subtree has just one node, and it's balanced
      delete tbd;
      lower=true; // balances will change on path to root
```

```
    } else { // both subtrees are non-empty => find inorder successor s,
      // copy it's value to root, and delete the value in the right subtree
      // note: recursion impossible, inorder successor has 1 child or none
      avlPtr iosucc=root->right; // is not NULL
      while (iosucc->left!=NULL) {
        iosucc=iosucc->left;
      }
      root->value=iosucc->value;
      cout << "Copy value of inorder successor " << iosucc->value
           << " into the node " << value << ", and delete the original "
           << iosucc->value << " in the right subtree" << endl;
      avlDeleteH(root->right,iosucc->value,lower,deleted);
      // this way, subtree was traversed twice (and maybe copying the value
      // is expensive in case it's a huge struct), but traversing twice
      // isn't much difference compared to having root's value at hand during
      // recursion (and we'd have to implement another delete routine,
      // because we don't know inorder successor's value in advance)
      // only drawback: if there are other pointers to avl tree's nodes
      // those pointers will point to the wrong node (due to copying the value)
      if (lower) { // copied from above avlDeleteH(root->right,...
        if (root->balance==unbalancedleft) {
          if (root->left->balance==balanced) {
            lower=false;
          } // else { // lower is still true after rotation
          cout << "Rotation necessary @ " << root->value
               << " -- subtree:" << root;
          avlRotate(root);
        } else if (root->balance==balanced) {
          root->balance=unbalancedleft;
          lower=false;
          cout << "Node " << root->value
               << " is now unbalanced to the left" << endl;
        } else { // root->balance==unbalancedright
          root->balance=balanced;
          cout << "Node " << root->value << " is now balanced" << endl;
          // lower is still true
        }
      }
    }
    deleted=true; // yes, we deleted an element
  }
}

////////////////////////////////////////////////////////////////////////////
bool avlDelete(avlPtr & root, const DataType value) {
////////////////////////////////////////////////////////////////////////////
  bool lower = false; bool deleted = false;
  avlDeleteH(root,value,lower,deleted);
  return deleted;
}
```

```
305 ////////////////////////////////////////////////////////////////////////////
    void delTree(const avlPtr root) {
    ////////////////////////////////////////////////////////////////////////////
      if (root!=NULL) {
        delTree(root->left);
310     delTree(root->right);
        delete root;
      }
    }

315 ////////////////////////////////////////////////////////////////////////////
    bstPtr avl2bst(const avlPtr root) {
    ////////////////////////////////////////////////////////////////////////////
      if (root==NULL) {
        return NULL;
320   } else {
        bstPtr cp=new bsTree(root->value,avl2bst(root->left),avl2bst(root->right));
        //     cp->value=root->value;
        //     cp->left=avl2bst(root->left);
        //     cp->right=avl2bst(root->right);
325     return cp;
      }
    }

    ////////////////////////////////////////////////////////////////////////////
330 bstPtr avlbal2bst(const avlPtr root) {
    ////////////////////////////////////////////////////////////////////////////
      if (root==NULL) {
        return NULL;
      } else {
335     bstPtr cp =
        new bsTree(root->balance-1,avlbal2bst(root->left),avlbal2bst(root->right));
        //     cp->value=root->balance-1; // "conversion" of enum type to a number
        //     cp->left=avlbal2bst(root->left);
        //     cp->right=avlbal2bst(root->right);
340     return cp;
      }
    }
```

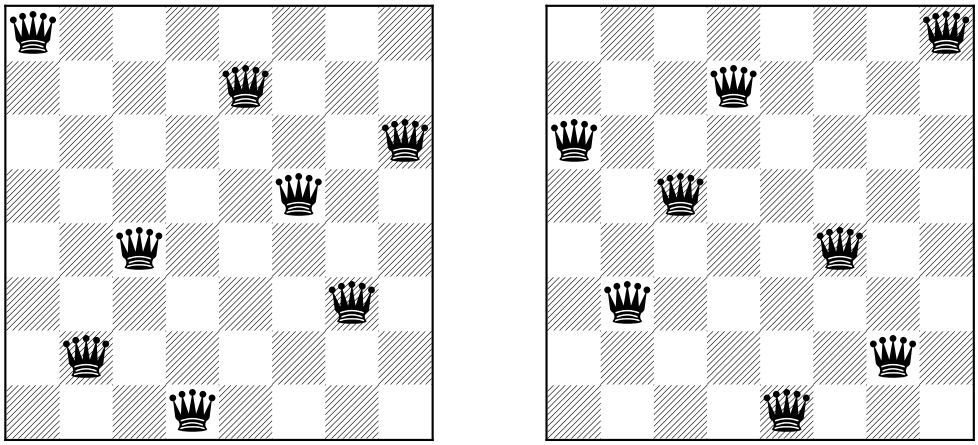
```
////////////////////////////////////////////////////////////////////////////
void avlInsertFiboOfHeight(avlPtr & root, const int height) {              //
// inserts elements of a fibonacci tree of <height>, <root> will be emptied //
////////////////////////////////////////////////////////////////////////////
  delTree(root);
  if (height>1) {
    int *fibo=new int[height+1]; fibo[0]=1; fibo[1]=1;
    // = root's value on level [.] // fibo[0]=1 only needed 10 lines below ...
    for(int i=2; i<=height; i++) { fibo[i]=fibo[i-1]+fibo[i-2]; }
    // recursive insertion isn't possible (due to induced rotations), use queue
    const int nrOfNodesInFiboTree = fibo[height]+fibo[height-1]-1;
    int *fiborder=new int[nrOfNodesInFiboTree]; //order of nodes to be inserted
    int *fibheight=new int[nrOfNodesInFiboTree];//heights of trees in *fiborder
    int front,next; // "queue" index - this pseudo queue is good enough
    // in a fibo tree root and it's left descendants are fibo nums (1,2,3,5,..)
    // left subtree's root has value -fibnr[height-2], right +fibnr[height-2]
    // we can calculate the heights, values and insert them in parallel
    fibheight[0]=height;
    fiborder[0]=fibo[height];
    avlInsert(root,fiborder[0]);
    front=0; next=1;
    while (next<nrOfNodesInFiboTree) {
      // insert "subtrees" of height .-1 and .-2
      if (fibheight[front]>1) { // left subtree exists
        fibheight[next]=fibheight[front]-1;
        fiborder[next]=fiborder[front]-fibo[fibheight[front]-2];//fibo[0]needed
        avlInsert(root,fiborder[next]);
        next++;
      }
      if (fibheight[front]>2) { // right subtree exists
        fibheight[next]=fibheight[front]-2;
        fiborder[next]=fiborder[front]+fibo[fibheight[next]]; // +f[h[front]-2]
        avlInsert(root,fiborder[next]);
        next++;
      }
      front++;
    }
    delete fibo; delete fiborder; delete fibheight;
  } else if (height==1) {
    root=new avlTree; root->value=1;
    root->left=root->right=NULL;
    root->balance=balanced;
  } // else height<1, root=NULL
}
```

# B Begleitmaterialien und Aufgaben für das Labor

**Vorbemerkungen:** Im zweiten Semester behandeln wir viele Standard-Datenstrukturen und -Algorithmen. Der Schwerpunkt im Labor soll dabei auf der experimentellen Untersuchung liegen. Wir werden hier immer wieder überprüfen, inwieweit theoretische Abschätzungen sich in der Praxis wiederspiegeln. Zum Teil wird dies durch Zählen von „Schritten“, zum Teil durch Zeitmessungen passieren.

## B.1 Iteration oder Rekursion? – das $n$-Damen-Problem

Das 8-Damen-Problem ist ein Klassiker unter den Aufgaben zur systematischen Suche in einem Lösungsraum – die Aufgabe besteht darin, 8 Damen so auf einem Schachbrett zu verteilen, dass diese sich gegenseitig nicht bedrohen (d. h., keine zwei Damen stehen in derselben Zeile, Spalte oder Diagonalen). Zwei mögliche Lösungen zeigt das folgende Bild (dies sind die Lösungen, die mit den üblichen Ansätzen als erste und letzte gefunden werden):



Ein mögliches Vorgehen von Hand besteht darin, systematisch alle möglichen Kombinationen durchzuprobieren. Wir werden dabei gleich in der ersten Programmversion versuchen, offensichtlich falsche Aufstellungen zu ignorieren, indem in jeder Zeile genau eine Dame platziert wird.

Schreiben Sie zunächst ein kleines Programm, mit dem ein Schachbrett verwaltet werden kann und das alle Möglichkeiten auflistet, in jeder Zeile genau eine Dame zu platzieren (zunächst ohne dabei darauf zu achten, ob sich dabei zwei Damen möglicherweise bedrohen). Sinnvollerweise wird man hierbei zeilenweise vorgehen (platziere eine Dame in der ersten Zeile, für jede Möglichkeit platziere eine Dame in der zweiten Zeile, für jede dieser Kombinationen platziere eine Dame in der dritten Zeile usw.).

Fügen Sie nun eine Funktion hinzu, die vor dem Setzen einer Dame auf das Feld überprüft, ob dieses von einer der bereits gesetzten Damen bedroht wird – die Dame soll nur dann dorthin gesetzt werden, wenn das Feld noch frei ist. Wieviel Lösungen erhalten Sie?

Das Programm hat in der naiven Version ohne Rekursion (nur mit geschachtelten Schleifen) eine sehr einheitliche Struktur. Spätestens, wenn man das Programm erweitern will, sodass es auch auf größeren Schachbrettern funktioniert, ist das Vorgehen über Schleifen in dieser Form nicht mehr möglich. Schreiben Sie das Programm neu in einer rekursiven Fassung (die

Funktion wird hier u. A. einen Parameter für die aktuelle Zeile erwarten, in der als nächstes eine Dame gesetzt werden soll).

Anmerkung: es gibt auch einen iterativen Algorithmus, der sich aus der Herangehensweise bei der rekursiven Lösung ableiten lässt.

Führen Sie Zeitmessungen für größere $n$ durch und vergleichen Sie die Laufzeiten der rekursiven und iterativen Varianten.

Untersuchen Sie (neben der Verallgemeinerung auf andere Schachbrettgrößen) weitere Varianten des $n$-Damen-Problems:

- Bestimmen Sie die minimale Anzahl der Damen auf einem Schachbrett, sodass jedes Feld von einer Dame bedroht wird.
- Bestimmen Sie die Anzahl der Lösungen des $n$-Damen-Problems, wenn das Schachbrett zu einem Torus geschlossen wird (ein Torus kann man sich so vorstellen, dass der linke und rechte sowie der obere und untere Rand zusammengeklebt werden – in diesem Kontext sind nur die Diagonalen relevant, die sinngemäß am jeweils gegenüberliegenden Schachbrettrand fortgesetzt werden). Hinweis: für $n = 8$ hat diese Variante keine Lösung, probieren Sie z. B. $n = 5$, 7 oder 11.
- Verändern Sie die Zugmöglichkeiten der Figuren – erlauben Sie z. B., dass Damen sich zusätzlich wie Springer bewegen können.
- Erlauben Sie Hindernisse auf dem Schachbrett – z. B. lassen sich mit einem blockierten Feld auf einem $8 \times 8$-Schachbrett auch 9 Damen so verteilen, dass sich diese nicht gegenseitig bedrohen.
- Entwerfen Sie weitere Varianten.

## B.2 Das Münzproblem

siehe Seite 14

## B.3 Einfallspinsel – Automaten und Zeiger auf Funktionen spielerisch eingeführt

Der Begriff des Automaten taucht in der Informatik in sehr verschiedenen Gebieten auf. Zentrale Elemente eines Automaten sind Zustände und Zustandsübergänge (auch Transitionen genannt). Die Menge der möglichen Zustandsübergänge bestimmt dabei das Verhalten des Automaten. Wir werden im Verlauf dieses Abschnitts einen Automaten programmieren.[21]

Zeiger auf primitive Datentypen, Arrays und Verbunde (`struct`) haben wir bereits kennengelernt, es gibt darüberhinaus auch Zeiger auf Funktionen! Wir führen dies am Beispiel eines einfachen Sortierverfahrens (das schon bekannte Sortieren durch Minimumsuche) für einen selbstdefinierten Datentyp für Brüche ein.

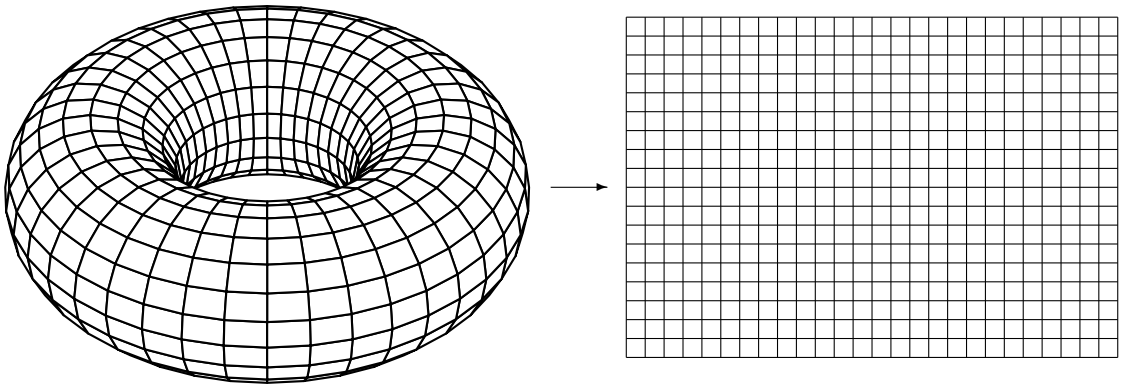
[21] Eine Übersicht über verschiedene Typen von Automaten in der Informatik liefert z. B. `http://de.wikipedia.org/wiki/Automat_(Informatik)`

Sortieren von Brüchen – Beispiel für Zeiger auf Funktionen

```
#include <iostream>
using namespace std;

struct Bruch { int z,n; };

ostream& operator << (ostream & os, const Bruch & a) {
  os << '(' << a.z << '/' << a.n << ')';
  return os;
}

bool kleiner(const Bruch & a, const Bruch & b)
{ return a.z*b.n<b.z*a.n; }

bool groesser(const Bruch & a, const Bruch & b)
{ return a.z*b.n>b.z*a.n; }

bool nachkommakleiner(const Bruch & a, const Bruch & b)
{ return (a.z%a.n)*b.n<(b.z%b.n)*a.n; }

void tausche(Bruch & a, Bruch & b) { Bruch h=a; a=b; b=h; }

void MinimumSort(Bruch tosort[], const int l, const int r,
                 bool (*aLinksVonB)(const Bruch & a, const Bruch & b))
{
  int mindex, i, j;
  for(i=l;i<r;i++)
  {
    mindex=i;
    for(j=i+1;j<=r;j++) if (aLinksVonB(tosort[j],tosort[mindex])) mindex=j;
    tausche(tosort[i],tosort[mindex]);
  }
}

#define N 7

int main(void)
{
  Bruch a[N]={{4,5}, {7,6}, {1,12}, {23,5}, {14,11}, {1,2}, {4,1}};
  int i;
  for(i=0;i<N;i++) cout << ' ' << a[i]; cout << endl;
  MinimumSort(a,0,N-1,kleiner);
  for(i=0;i<N;i++) cout << ' ' << a[i]; cout << endl;
  MinimumSort(a,0,N-1,groesser);
  for(i=0;i<N;i++) cout << ' ' << a[i]; cout << endl;
  MinimumSort(a,0,N-1,nachkommakleiner);
  for(i=0;i<N;i++) cout << ' ' << a[i]; cout << endl;

  return 0;
}
```

Das Programm implementiert drei Funktionen zum Vergleich von je zwei Brüchen – „kleiner“, „größer“ sowie „kleiner“, wobei hier nur der Nachkommaanteil berücksichtigt wird. Zum Sortieren muss man lediglich entscheiden können, ob für zwei Elemente das erste links vom zweiten

stehen muss oder umgekehrt. Das Entscheidungskriterium – eine der oben implementierten Funktionen – wird dazu als Argument für das Sortierverfahren übergeben.



**Einfallspinsel – ein Spiel:** Die Welt der Einfallspinsel ist ein Gitter auf einem Torus – vereinfacht dargestellt: ein Rechteck-Gitter, bei dem der rechte und linke Rand sowie der obere und untere Rand zusammengeklebt sind (verlässt man das Gitter auf der rechten Seite, tritt man in derselben Zeile links wieder ein, analog mit oben und unten).

Die Felder auf dem Torus sollen eingefärbt werden. Ziel ist, am Ende möglichst viele Felder mit seiner Farbe eingefärbt zu haben. Das Spiel ist nach einer vorher festgelegten Rundenanzahl (hier 100) zu Ende. Eine Runde besteht darin, dass jeder der zu Beginn der Runde vorhandenen Pinsel einen Zug durchführt.

Zu Beginn wird für jeden Mitspieler je ein Pinsel an jeweils zufälliger Position auf dem Gitter platziert, die genaue Größe des Torus ist dabei den Pinseln nicht unbedingt bekannt (hier werden 26×23 Felder verwendet). Jedes Feld kann nur von einem Pinsel belegt werden. Ein Pinsel kann nur die vier direkt benachbarten Felder und die vier diagonal benachbarten um sich herum wahrnehmen und muss daraufhin entscheiden, was er tut (zusätzlich weiß er noch, was als letzter Zug durchgeführt wurde). Er hat folgende Möglichkeiten:

- Ist das Feld, auf dem er steht, noch unmarkiert, so kann er es (mit seiner Farbe) einfärben und stehenbleiben oder danach noch auf eines der vier direkt benachbarten Felder wechseln. Ebenfalls kann er erst ziehen und dann (optional) das erreichte Feld markieren. (Züge: `MARK`, `MV_NORTH`, `MV_NORTH_MARK`, `MARK_MV_NORTH`, `MV_EAST`, `MV_EAST_MARK`, `MARK_MV_EAST`, `MV_SOUTH`, `MV_SOUTH_MARK`, `MARK_MV_SOUTH`, `MV_WEST`, `MV_WEST_MARK`, `MARK_MV_WEST`)
- Auf eines der vier diagonal liegenden Felder wechseln oder versuchen, sich zwei Felder in horizontaler oder vertikaler Richtung zu bewegen (jeweils ohne dass dieses Zielfeld noch in dieser Runde markiert werden kann). (Züge: `MV_NORTH_WEST`, `MV_NORTH_NORTH`, `MV_NORTH_EAST`, `MV_EAST_EAST`, `MV_SOUTH_EAST`, `MV_SOUTH_SOUTH`, `MV_SOUTH_WEST`, `MV_WEST_WEST`)

- Beim Versuch auf ein schon besetztes Feld zu wechseln, verfällt der Zug. (Rückmeldung: `FAILED` oder `PARTIALLY_FAILED`, wenn bei kombinierten Zügen mit Bewegung und Markierung erst der zweite Teil fehlschlug)
- Sich teilen, sodass danach zwei Pinsel unabhängig voneinander weiterpinseln können. Das Teilen ist jedoch aufwendig und benötigt vier Runden – in den ersten drei Runden setzt der Pinsel aus, in der vierten Runde wird auf einem der vier direkt benachbarten Felder (zufällig gewählt) ein neuer Pinsel erstellt. Sind in dieser Runde alle vier direkt benachbarten Felder besetzt, so schlägt das Teilen fehl und es verbleibt lediglich der alte Pinsel auf seinem Feld. (Zug: `DIVIDE`)
- Verwandlung in einen Superpinsel. Dieser erlaubt das Umfärben eines schon (vom Gegner) markierten Feldes. Auch bei diesem Umwandlungsprozess setzt der Pinsel drei Runden aus und wird in der vierten Runde gewandelt. Achtung: die Superpinseleigenschaft geht bei einer später folgenden Teilung wieder verloren! Bis dahin, können aber in jeder Runde gegnerische Felder umgefärbt werden. (Zug: `MORPH`)
- Falls ein Pinsel meint, das es geschickt wäre, nichts zu tun, so ist auch dieses möglich. (Zug: `NOTHING`)

Ist der Pinsel an der Reihe, so bekommt er nur die für ihn sichtbaren Felder mitgeteilt (für jedes Feld dessen Markierung sowie die Belegung (Markierungen: `EMPTY_NOMARK`, `EMPTY_FRIENDMARK`, `EMPTY_RIVALMARK`, `FRIEND_NOMARK`, `FRIEND_FRIENDMARK`, `FRIEND_RIVALMARK`, `RIVAL_NOMARK`, `RIVAL_FRIENDMARK`, `RIVAL_RIVALMARK`) und muss sich daraufhin einfallen lassen, was er zu tun gedenkt (deshalb auch der Name „Einfallspinsel“). Ist er wieder an der Reihe, so bekommt er (neben den sichtbaren Feldern) zusätzlich noch mitgeteilt, was sein letzter Zug war[22].

Jeder Mitspieler liefert eine Headerdatei, die exakt folgenden Aufbau haben soll:

Aufbau der Headerdatei eines Mitspielers

```
#ifndef MNR1234567_H
#define MNR1234567_H

#include "einfallspinsel_defs.h" // beinhaltet nur die Typdefinitionen

ACTtype mnr1234567(PBRtype,CONtype,CONtype,CONtype,CONtype,CONtype,
    CONtype,CONtype,CONtype,CONtype,ACTtype,void*&);

#endif
```

Und die Implementierung seiner Funktion, wir betrachten hier als Beispiel einen sehr einfachen Pinsel, der – wenn möglich – markiert und nach Norden läuft und nur bei Bedarf nach Osten ausweicht:

Implementierung eines Mitspielers

```
#include "mnr1234567.h"

ACTtype mnr1234567(PBRtype mytype, CONtype north_west, CONtype north,
    CONtype north_east, CONtype west, CONtype here, CONtype east,
```

[22] In der ersten Runde ist hier als letzter Zug `ROUND_ZERO` vermerkt, bei Teilung ist der letzte Zug des Elters `DIVIDE`, der des Kindes `ROUND_ZERO`; war der letzte Zug kein bekannter Befehl, so lautet die Rückmeldung `INVALID`.

```
    CONtype south_west , CONtype south , CONtype south_east ,
    ACTtype lastact , void* & dataptr )
{
  if ( north ==EMPTY_NOMARK) return MV_NORTH_MARK;
  // die eigentliche Bewegung wird vom Programm ausgeführt ,
  // die Funktion muss nur zurückgeben , was getan werden soll
  else if ( east ==EMPTY_NOMARK) return MV_EAST_MARK;
  else if ( south ==EMPTY_NOMARK) return MV_SOUTH_MARK;
  else if ( west ==EMPTY_NOMARK) return MV_WEST_MARK;
  else return MV_EAST;
  // falls da schon wer steht , Pech gehabt ( schlägt dann fehl )
}
```

Dieser einfache Pinsel benutzt nicht die Möglichkeit, den Zug abhängig vom zuletzt durchgeführten Zug oder von seinem Typ zu wählen.

In diesem Spiel können wir zusätzlich noch die Verwendung **untypisierter Pointer** kennenlernen. Jeder Pinsel hat einen Zeiger auf die von ihm verwalteten Daten (der Zeiger ist mit `NULL` initialisiert). Um hier keine Vorgaben über Datentypen machen zu müssen, ist der Pointer vom Datentyp **void***. Im Beispiel wird der Pointer als Referenz (&) übergeben, sodass eine Änderung des Pointers (also der Adresse, auf die der Pointer zeigt) in der Funktion möglich ist. Betrachten wir als Beispiel damit die Möglichkeit, die Anzahl der Runden seit Entstehung des Pinsels zu zählen ( dataptr ist dabei der als Referenz übergebene Zeiger):

Verwendung des dataptr

```
  if ( dataptr ==NULL)
  {
    int* a=new int; // Speicher anfordern für die int-Variable
    // muss dynamisch geschehen , weil sonst die Variable mit Verlassen der
    // Funktion gelöscht würde
    dataptr =(void*)a; // "Kopieren" des Pointers auf die Referenzvariable
    // (mit Umwandeln des Pointer-Typs)
    *a=1;
  }
  else
  {
    int* a=(int*) dataptr ; // Speicher wurde schon mal angefordert
    (*a)++; // Anzahl der bisherigen Aufrufe hochzählen
  }
```

Es können hier beliebige Datentypen verwendet werden, egal ob einfache Datentypen, Arrays oder selbstdefinierte Strukturen. Wichtig ist lediglich, den Speicher für diese mittels **new** anzufordern (und ggf. mit **delete** zu löschen).

Für das Spiel ist die Verwendung dieses Mechanismus nicht zwingend notwendig, es erlaubt aber Daten zu sammeln und zwischen den Pinseln auszutauschen (bei Teilung wird der Zeiger vererbt, soll ein geteilter Pinsel einen eigenen, unabhängigen Speicherbereich haben, so kann er dies durch dataptr !=NULL && lastact ==ROUND_ZERO erkennen – Quellcode anschauen und selbst experimentieren).

Ein Wermutstropfen am Ende ... wie fast alles ist auch das Leben eines Pinsels endlich – mit einer geringen Wahrscheinlichkeit stirbt dieser zu Beginn einer Runde und wird vom Spielfeld entfernt. Um dem Mitspieler zu erlauben, dies in den ggf. selbst verwalteten Daten zu berücksichtigen, wird der Pinsel ein weiteres Mal aufgerufen – er erhält als Information über den

vorherigen Zug den Wert `DEATH`. Der Rückgabewert, den der Pinsel an das Hauptprogramm liefert, wird ignoriert und der Pinsel danach entfernt.

In der aktuellen Fassung des Spiels ist zum Starten und Spielen noch etwas Handarbeit nötig. Die nötigen Schritte sind im Quellcode vermerkt, der auf der Vorlesungsseite zur Verfügung gestellt wird.

## B.4 Einfügen und Löschen in binären Suchbäumen

siehe Seite 24

## B.5 Sortieren mit AVL-Bäumen

Hintergrund: Wir haben in Abschnitt 1.3.3 erfahren, dass die mittlere Suchdauer (identisch mit der mittleren Dauer zum Einfügen eines Elementes) in einem binären Suchbaum logarithmisch beschränkt ist (genauer: in der Summe werden beim Einfügen der Elemente in einen binären Suchbaum (als Erwartungswert) $1.386 \cdot n \cdot \log n - 1.846 \cdot n$ Ebenen durchlaufen) – der schlechteste Fall kann aber quadratisch werden (genau: $n(n+1)/2$ Ebenen).

AVL-Bäume wurden eingeführt, um die Such- und Einfügezeiten nicht nur im Mittel, sondern auch im Worst-Case logarithmisch zu beschränken.

Aufgabe: Nachdem in Abschnitt B.4 für zufällig erzeugte binäre Suchbäume die Summe der Level (also die Anzahl der betrachteten Ebenen beim Einfügen der Elemente in einen anfangs leeren binären Suchbaum) untersucht haben, soll diese Anzahl mit derjenigen bei der Verwendung von AVL-Bäumen verglichen werden (das zugehörige Modul ist auf der Vorlesungsseite verfügbar). Welche Konstante können Sie für den Term vor dem $n \cdot \log n$ ermitteln? Hinterfragen Sie auch hier, wieviel Durchläufe des Experiments – ggf. abhängig von der Anzahl der Knoten – nötig sind, um ein stabiles Ergebnis zu erhalten.

Die Implementierung von AVL-Bäumen mit der notwendigen Verwaltung der Höhenbalancen ist aufwendiger als bei einfachen binären Suchbäumen. Wir wollen deshalb auch die realen Laufzeiten vergleichen. Hierzu finden Sie auf der Vorlesungsseite ein Modul `stoppuhr(.h/.cpp)`, mit dem eine Zeitmessung unter Windows und Linux möglich ist.

Zeitmessungen sind auf heutigen Rechnersystemen nicht unproblematisch, da hier viele Aspekte der Hardwarearchitektur eine Rolle spielen (neben der Taktfrequenz z. B. die Anzahl der Prozessorkerne und die Größe des Caches), aber auch bei gleicher Hardware der Zufall eine Rolle spielt (bei dynamischer Speicherverwaltung werden einzelne Knoten eines Baumes mal an tendenziell günstigen, mal an ungünstigen Stellen im Speicher abgelegt (ungünstig in dem Sinne, dass bei Zugriffen auf mehreren Knoten diese in einem zusammenhängenden Speicherbereich oder verteilt liegen können – diese Effekte spielen beim Cache eine große Rolle).

Aufgabe: Ermittlung der Schwankung bei Zeitmessung bei identischen Bäumen: Die Initialisierung des Zufallszahlengenerators bietet eine Möglichkeit Experimente wiederholbar zu machen. Es bietet sich also an, vor Ermittlung einer einzufügenden Permutation, den Zufallszahlengenerator jeweils neu zu initialisieren (z. B. für das erste Experiment mit 1, für das zweite mit 2, usw.). Fügen Sie dieselben Zahlen in derselben Reihenfolge (Initialisierung des Zufallszahlengenerators jeweils mit derselben Zahl) immer wieder in einem jeweils zu Beginn leeren binären Suchbaum ein und messen Sie die benötigte Zeit. Wählen Sie die Anzahl der Elemente so groß, dass Ihnen eine Zeitmessung sinnvoll möglich erscheint. Wiederholen Sie auch hier das Experiment ausreichend oft und ermitteln Sie die Streuung der Ergebnisse.

Aufgabe: zurück zur ersten Teilaufgabe – messen Sie nun für die Bäume die benötigte Zeit und vergleichen Sie diese mit dem Ergebnis, was durch Betrachtung der Summe der Level ermittelt wurde.

## B.6 Suche nach dem $k$-kleinsten Element

Hintergrund: Wir haben auf Seite 45 erfahren, dass der mittlere Aufwand zum Sortieren mit Quicksort $1.386 \cdot n \cdot \log n - 0.846 \cdot n$ Schlüsselvergleiche benötigt (Indexvergleiche werden dabei nicht gezählt). Im schlechtesten Fall kann der Aufwand aber quadratisch werden. Manchmal möchte man nur das $k$-kleinste Element bestimmen (z. B. den Median). Dazu muss beim Quicksort-Algorithmus nur die eine Hälfte des Feldes sortiert werden (die, in der das gesuchte Element liegt); die Reihenfolge der Elemente in der anderen Hälfte spielt keine Rolle.

Ein Beispiel: Sei das Feld zu Beginn mit den 15 Zahlen

3 14 15 92 6 5 35 8 97 93 2 38 4 62 64

belegt. Ziel sei hier die Bestimmung des Medians (dies ist das Element, das in der sortierten Folge in der Mitte steht (bei gerader Elementeanzahl gibt es zwei Mediane, man betrachtet in der Regel den linken), also den am Index („linker Rand“+“rechter Rand“)/2 – in C-Arrays am Index N/2 (wenn N die Anzahl der Elemente im Array ist)). Der erste Quicksortschritt liefert (Pivot ist 8, Tausch 14↔4, 15↔2, 92↔8, der linke Zeiger wandert zur 35, der rechte zur 5)

3 4 2 8 6 5 || 35 92 97 93 15 38 14 62 64

Der Quicksort-Algorithmus würde nun rekursiv die ersten 6 und die hinteren 9 Elemente sortieren. Um den Median (das Element an Index 7) zu bestimmen, reicht es, die hintere Hälfte weiterzusortieren. Wir erhalten (Pivot 15, Tausch 35↔14, 92↔15, der linke Zeiger bleibt auf der 97, der rechte läuft bis zur 15)

3 4 2 8 6 5 || 14 15 || 97 93 92 38 35 62 64

Der Median liegt im mittleren Bereich (Indizes 6 und 7). Mit Pivot 14 gehen beide Zeiger auf die 14, nach Tausch mit sich selbst, steht der linke Zeiger auf der 15, der Bereich, der den Index 7 enthält, ist auf ein Element geschrumpft und somit die 15 als Median des obigen Feldes bestimmt.

Ihre Aufgaben: Programmieren Sie den Quicksortalgorithmus so um, dass er das $k$-te Element der sortierten Reihenfolge bestimmt ($k$ ist dann ein weiterer Parameter der Funktion – nennen wir sie „select“). Da nur ein rekursiver Aufruf nötig wäre, kann man diesen auch leicht umgehen, indem man eine while-Schleife programmiert und nach jedem Durchlauf den linken und rechten Rand anpasst, bis der noch relevante zu „sortierende“ Bereich auf (höchstens) ein Element geschrumpft ist.

Versuchen Sie die benötigte Anzahl der Schlüsselvergleiche zum Finden des Medians in Abhängigkeit von der Anzahl der Elemente im Array zu bestimmen (stellen Sie aufgrund der Ergebnisse für verschieden große Arrays eine Funktion auf).

Beachten Sie: Es findet bei den while-Schleifen immer ein Vergleich mehr statt als die Anzahl der Schleifendurchläufe beträgt. (Ein Vergleich wird benötigt, um festzustellen, dass die Schleife nicht mehr betreten wird!)

Empfehlung: die Korrektheit Ihres select-Algorithmus lässt sich leicht verifizieren, wenn Sie das Array mit einer Permutation initialisieren (Werte 0 bis N-1). Das $k$-te Element (wenn wir $k$ bei 0 beginnen zu zählen) der sortierten Folge hat dann den Wert $k$.

Versuchen Sie, Fälle zu konstruieren, die möglichst viele Schlüsselvergleiche benötigen, bevor der Median gefunden wird (wählen Sie hier ein festes N, z. B. N=7 oder N=100). Welche maximale Anzahl an Vergleichen haben Sie in Ihren Experimenten erreicht. Was können Sie sich als maximalen Wert aus theoretischen Überlegungen heraus vorstellen? (Sie können hierzu entweder zufällige Permutation wählen, bei N=7 könnte man alle Permutationen testen, oder sich einen Algorithmus ausdenken, der „schwierige Permutationen“ findet – seien Sie kreativ)

Vorschläge für weitere Untersuchungen: Für den Quicksortschritt gibt es viele verschiedene Implementierungen. Eine weit verbreitete tauscht zunächst das Pivot-Element an den rechten Rand, führt für den Rest des Feldes den Quicksortschritt wie in der Vorlesung gelernt aus und tauscht dann das Pivot-Element an den linken Rand der rechten Hälfte (somit steht das Pivot-Element nach dieser Quicksortschritt-Implementierung garantiert an der (bzgl. Sortierung) richtigen Position und braucht in der Rekursion nicht mehr berücksichtigt zu werden. Testen Sie, ob diese Implementierung des Quicksortschritts für die select-Funktion Vorteile gegenüber der oben bereits implementierten Variante hat (steht das Pivot-Element nach dem Tausch an seiner korrekten Position an dem gesuchten Index $k$, so kann bei dieser Variante in diesem Moment abgebrochen werden – es sind hier aber zusätzliche Tausch-Operationen notwendig, versuchen Sie diese Eigenschaften in Ihrem Vergleich zu berücksichtigen).

## B.7 Vergleich von Sortieralgorithmen

siehe Seite 52

## B.8 Projektplanung – eine Anwendung für Tiefensuche in Graphen und Kürzeste-Wege-Suche in azyklischen Graphen

Wir betrachten folgendes Programm [23]

Projektplanung

```
#include <time.h>
#include <cstdlib>
#include <iostream>
using namespace std;

struct intList { int idx; intList *next; };

void dfs(const int i, int * const dfsNum, int * const dfsFin,
         int & dfscnt, int & fincnt, const intList * const * const mussVor)
{ // Schema: falls Knoten i noch nicht markiert ist,
  // markiere Knoten i und führe die Tiefensuche von allen Nachfolgern aus aus.
  if (dfsNum[i]==-1) { // noch nicht markiert
    dfsNum[i]=dfscnt++;
    const intList *job = mussVor[i];
    while (job!=NULL) {
      dfs(job->idx,dfsNum,dfsFin,dfscnt,fincnt,mussVor);
      job=job->next;
```

[23] siehe `http://inf4dhbw.warumauch.net/Jahrgang2009/dfsanwdg_halb.cpp`

```
    }
    dfsFin[i]=fincnt--;
  }
}


int main(void)
{
  srand((unsigned) time(NULL)); // Initialisierung des Zufallszahlengenerators

#define ANZJOBS 7
#define MAXDAUER 9
#define ANZABHG ANZJOBS

  // zufällige Probleminstanz erzeugen

  int i,a,b;
  intList *job;

  int dauer[ANZJOBS]; // gibt die Dauer der Arbeit i an
  for(i=0;i<ANZJOBS;i++) dauer[i]=rand()%MAXDAUER +1;

  intList *mussVor[ANZJOBS]; // gibt für jeden Knoten i
  // eine Liste der Indizes j_1,...,j_(k_i) an, sodass Arbeit i vor Arbeit j_x
  // abgeschlossen sein muss
  for(i=0;i<ANZJOBS;i++) mussVor[i]=NULL;
  for(i=0;i<ANZABHG;i++)
  {
    a=rand()%ANZJOBS; b=(a +1 +(rand()%(ANZJOBS-1)))%ANZJOBS; // 2 zuf. Jobs a!=b
    // a soll vor b ausgeführt werden
    job=new intList; job->next=mussVor[a]; job->idx=b; mussVor[a]=job;
    // b an den Anfang vor die bisherige Liste von a hängen
  }

  cout << "Test-Instanz fuer einen zu erstellenden Arbeitsplan" << endl;
  for(i=0;i<ANZJOBS;i++)
    cout << "Arbeit " << i << " dauert "
         << dauer[i] << " Zeiteinheiten." << endl;
  for(i=0;i<ANZJOBS;i++)
  {
    job=mussVor[i];
    while(job!=NULL)
    {
      cout << "Arbeit " << i
           << " muss vor Arbeit " << job->idx << " erfolgen." << endl;
      job=job->next;
    }
  }

  // durch Tiefensuche feststellen, ob die Abhaengigkeiten zyklisch sind und
  // - falls nicht - Bearbeitungsreihenfolge für den folgenden Algorithmus
  // bestimmen (dies hat zunächst nur indirekt mit der späteren realen Abfolge
  // der Arbeiten zu tun, die gewünschten Zeitpunkte lassen sich so deutlich
  // schneller und einfacher berechnen)
  int dfsNum[ANZJOBS]; for(i=0;i<ANZJOBS;i++) dfsNum[i]=-1; // -1 = noch nicht besucht
```

```
    int dfsFin[ANZJOBS]; for(i=0;i<ANZJOBS;i++) dfsFin[i]=-1; // -1 = noch nicht fertig
                                                              // Init nicht noetig
75  int dfscnt=0, fincnt=ANZJOBS-1; // nächste zu vergebene dfs/fin-Nummer
    for(i=0;i<ANZJOBS;i++) dfs(i,dfsNum,dfsFin,dfscnt,fincnt,mussVor);
    // DIESE ZEILE ist der ganze Zauber!

    // keine Zyklische Abhaengigkeiten vorhanden? (genau dann, wenn fuer jede
80  // Abhaengigkeit "a vor b" gilt: dfsFin[a]<dfsFin[b]
    for(i=0;i<ANZJOBS;i++)
    {
      job=mussVor[i];
      while(job!=NULL)
85    {
        if(dfsFin[i]>dfsFin[job->idx])
          cout << "Abhaengigkeit " << i
               << " vor " << job->idx << " kann nicht beruecksichtigt werden." << endl;
        job=job->next;
90    }
    }

    int azykl[ANZJOBS]; for(i=0;i<ANZJOBS;i++) azykl[dfsFin[i]]=i;
    cout << "Bearbeitungsreihenfolge: ";
95  for(i=0;i<ANZJOBS-1;i++) cout << azykl[i] << ", ";
    cout << azykl[ANZJOBS-1] << "." << endl;

    int fruehBeg[ANZJOBS], fruehEnd[ANZJOBS], spaetBeg[ANZJOBS], spaetEnd[ANZJOBS];
    for(i=0;i<ANZJOBS;i++) fruehBeg[i]=fruehEnd[i]=0;
100 int lastexit=0;
    for(i=0;i<ANZJOBS;i++)
    {
      fruehEnd[azykl[i]]=fruehBeg[azykl[i]]+dauer[azykl[i]];
      if(fruehEnd[azykl[i]]>lastexit) lastexit=fruehEnd[azykl[i]];
105   job=mussVor[azykl[i]];
      while(job!=NULL)
      {
        if(dfsFin[azykl[i]]>dfsFin[job->idx])
          cout << "Abhaengigkeit " << azykl[i]
110            << " vor " << job->idx << " wird nicht beruecksichtigt." << endl;
        else
          if(fruehBeg[job->idx]<fruehEnd[azykl[i]]) fruehBeg[job->idx]=fruehEnd[azykl[i]];
        job=job->next;
      }
115 }
    for(i=0;i<ANZJOBS;i++)
      cout << "Arbeit " << i << " beginnt fruehestens um " << fruehBeg[i]
           << " und endet fruehestens um " << fruehEnd[i] << '.' << endl;
    cout << "Projektende nicht vor " << lastexit << "." << endl;
120
    return 0;
}
```

---

Es soll hier zunächst das Programm nachvollzogen werden

- Erstellung einer zufälligen Probleminstanz für Projektplan

- Bearbeitungsreihenfolge mittels Tiefensuche erstellen
- früheste Start- und Endzeitpunkte für jede Arbeit berechnen

Danach soll das Programm erweitert werden

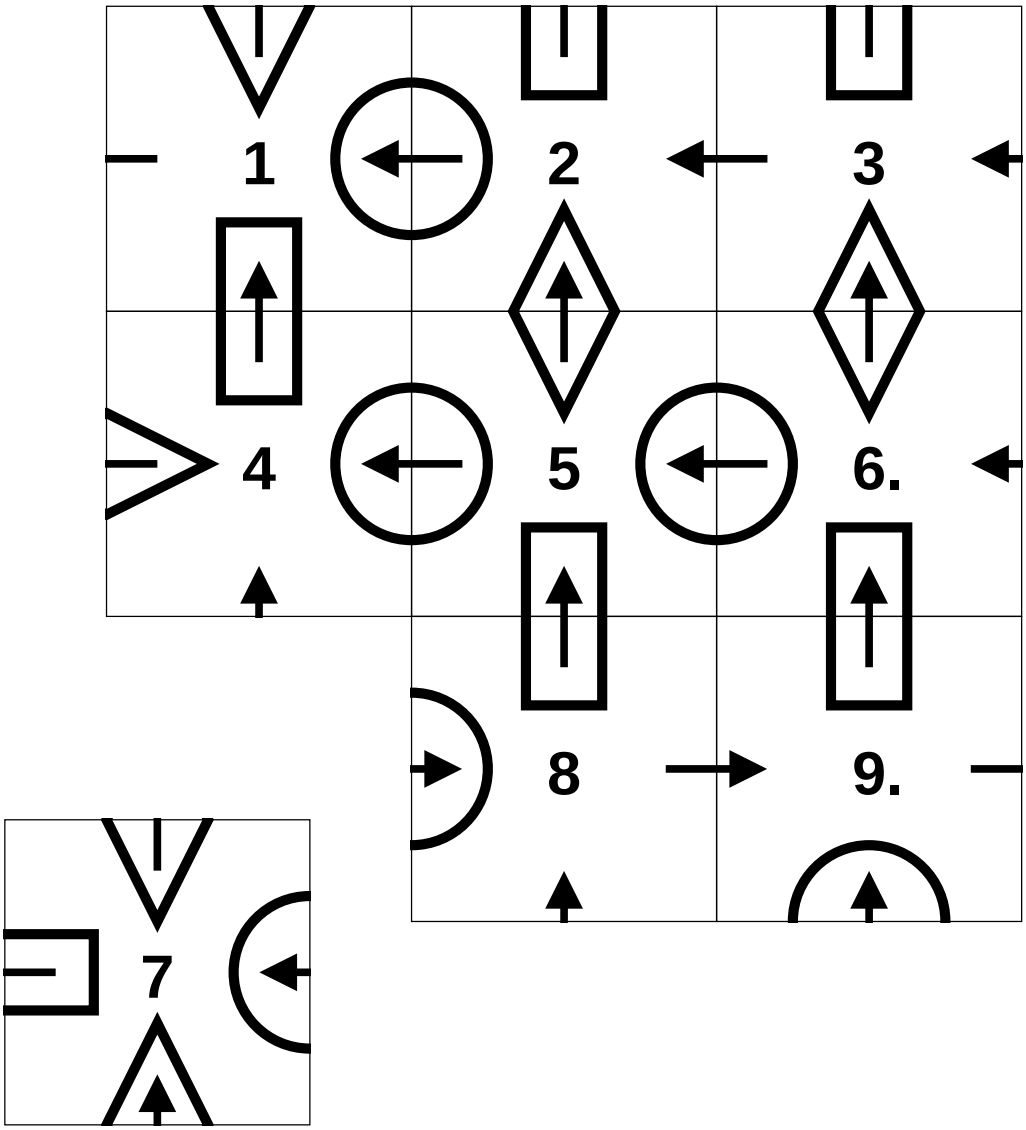
- Berechnung der spätestmöglichen Start- und Endzeitpunkte, sodass sich das Projekt nicht verzögert
- Bestimmung der kritischen Arbeiten, die sich nicht verzögern dürfen
- Bestimmung der Puffer für jede Arbeit (unter der Annahme, dass die anderen Arbeiten sich nicht verzögern)

# C Zusätzliche Übungsaufgaben

**Programmieraufgabe zur Wiederholung von Listen:** (Aufwand etwa 2–3 Stunden) Bei Abzählreimen („Ene mene mu und raus bist du“) wird immer der Reihe nach gezählt und (bei $x$ Silben) jeweils die $x$-te Person aus dem Kreis genommen, bis nur noch eine Person übrig ist. Bei diesem Verfahren ist die verbleibende Person eindeutig durch die Anzahl $n$ der Personen zu Beginn und die Anzahl $x$ der Silben im Abzählreim gegeben. Schreiben Sie ein Programm, dass für gegebene $n$ und $x$ zunächst eine Ringliste mit den Elementen 1 bis $n$ initialisiert und dann solange im Kreis das jeweils $x$-te Element entfernt, bis nur noch eines übrig ist. Untersuchen Sie, das wievielte Element für $x = 1$, $x = 2$, …, $x = n$ übrig bleibt. Welche Elemente kommen am häufigsten vor?

In einer zweiten Variante wollen wir nun noch etwas Zufall hinzufügen: statt das $x$ konstant zu halten, wollen wir dieses bei jedem Weiterzählen zufällig zwischen 1 und 6 wählen. Untersuchen Sie für $n = 10$ durch ausreichend häufige Wiederholung, ob es Positionen gibt, an denen ein Element mit höherer oder geringerer Wahrscheinlichkeit übrig bleibt. Variieren Sie die die Werte für $n$ und den Bereich, aus dem die Zufallszahlen kommen. Erweitern Sie die Aufgabe mit eigenen Fragestellungen.

**Rekursives Lösen von Legepuzzlen:** An manchen Knobelaufgaben kann man nur verzweifeln – der Computer ist ideal geeignet, solche Aufgaben durch Probieren aller Möglichkeiten zu lösen. Hier eine weitere Aufgabenstellung, die etwas Überlegung bei der Programmiervorbereitung benötigt. Bei folgendem Puzzle sollen die 9 Teile zu einem $3 \times 3$-Quadrat zusammengefügt werden, sodass an jeder Kante eine vollständige Figur sowie ein Pfeil entsteht. Schreiben Sie ein entsprechendes Programm und bestimmen Sie die Anzahl der möglichen Lösungen. Überlegen Sie sich, wie Sie verhindern können, dass bis auf Drehung identische Lösungen mehrfach ausgegeben werden.




**Programmieraufgabe für Greedy- und Backtracking-Strategien:** Das Zahlenspiel Sudoku hat in den letzten Jahren außerordentliche Beliebtheit erlangt, in vielen Zeitungen werden regelmäßig Rätsel abgedruckt. Auf einem 9-mal-9-Spielfeld sollen in jede Zeile, jede Spalte sowie in jedes kleine 3-mal-3-Quadrat die Zahlen von 1 bis 9 so eingetragen werden, dass in keiner Zeile, Spalte oder Quadrat eine Zahl doppelt vorkommt. Wir wollen hier zunächst eine Hilfestellung für das Lösen der Rätsel programmieren und in einem zweiten Schritt selbst Sudoku-Rätsel erstellen. Hier eine mögliche Vorgehensreihenfolge (fehlende Details selbst überlegen):

- Überlegen Sie sich zunächst eine geeignete Datenstruktur und programmieren Sie eine Ausgabe des Spielfelds auf dem Bildschirm (z. B. jeweils 3er-Blöcke durch Leerzeichen und Leerzeilen getrennt). (Aufwand höchstens 1 Stunde)
- Programmieren Sie die Eingabe von Rätseln (oder kodieren Sie einige Rätsel zum Testen fest in Ihr Programm). (Aufwand höchstens 1 Stunde)
- Es gibt zwei einfache Regeln, die man intuitiv beim Lösen von Sudokus anwendet:
  - Kann in einer Zeile, Spalte oder Quadrat eine Zahl nur noch an eine Position gesetzt werden, so muss sie dort eingetragen werden (auch wenn noch andere Zahlen an diese Position passen würden).
  - Kann an einer Position nur noch eine einzige Zahl eingetragen werden, so muss diese dort stehen (auch wenn die Zahl in dieser Zeile, Spalte oder Quadrat auch noch an anderen Positionen stehen könnte).

  Überlegen Sie sich, wie Sie für jedes Feld die noch möglichen Kandidaten verwalten. Überlegen Sie sich, wie Sie an Ihrer Datenstruktur erkennen können, ob eine dieser beiden Regeln anwendbar ist. Programmieren Sie entsprechende Funktionen aus. (Aufwand mit Testen etwa 2-3 Stunden)
- Schreiben Sie eine gierige Funktion, die solange eine der beiden Regeln anwendbar ist, die entsprechende Zahl an der richtigen Position einträgt. (Viele der abgedruckten Rätsel werden Sie allein mit diesen beiden Regeln lösen können.) (zusätzlicher Aufwand etwa 1 Stunde)
- Bei manchen Rätseln kommt man damit nicht bis zum Ende durch – hier hilft manchmal raten weiter: Programmieren Sie mit Backtracking folgendes Vorgehen: wir nehmen uns ein beliebiges Feld und fügen dort eine der noch möglichen Zahlen ein. Danach kann man wieder versuchen die obigen Regeln anzuwenden. Erst, wenn dies wieder fehlschlägt, muss man erneut raten. Manchmal stellt man jedoch fest, dass das Rätsel noch nicht gelöst, aber auf manche Felder keine Zahl mehr ohne Verletzung der Sudoku-Eigenschaft eingetragen werden kann. Hier muss man mit Backtracking einen Schritt zurück. Programmieren Sie dieses rekursiv aus. So kann man nun jedes Rätsel in wenigen Sekunden automatisch lösen lassen. (zusätzlicher Aufwand 2-4 Stunden)
- Zum Erstellen von Rätseln können wir die schon programmierten Funktionen neu kombinieren: Ein mögliches Vorgehen: Wir beginnen mit einem leeren Feld und versuchen jeweils mit den 2 obigen Regeln Zahlen in das Rätsel einzutragen. Wenn dies nicht möglich ist, müssen wir jeweils weitere Zahlen zufällig hinzufügen (das Rätsel wird nachher nur aus diesen zufällig hinzugefügten Zahlen bestehen). Auch hier kann es passieren, dass die zufällig hinzugefügten Zahlen ein Rätsel unlösbar machen. In einer ersten Version können Sie den missglückten Versuch einfach verwerfen und von vorn beginnen (bei diesem Ansatz wird man etwa jedes zweite Rätsel verwerfen, die anderen ergeben lösbare Rätsel). (zusätzlicher Aufwand 2-4 Stunden)
- Man erhält so meist Rätsel mit etwa 25 bis 35 vorgegebenen Zahlen. Versuchen Sie nun noch, vorgegebene Zahlen zu entfernen, so dass das Rätsel trotzdem eindeutig lösbar bleibt (die eindeutige Lösbarkeit ist eine zusätzliche – oben noch nicht erwähnte – Sudoku-Eigenschaft). Dazu müssen Sie im Backtracking-Schritt nicht nur überprüfen, ob überhaupt eine Lösung möglich ist, sondern, ob es genau eine Lösung gibt (dies ist etwas aufwendiger). (zusätzlicher Aufwand 2-4 Stunden)

**Veständnisfragen zu AVL-Bäumen und optimalen Suchbäumen:**

- Welcher binäre Suchbaum entsteht, wenn man (in einen anfangs leeren Baum) die Elemente 3, 14, 15, 9, 26, 5, 35 in dieser Reihenfolge einfügt? Welcher AVL-Baum entsteht, wenn diese Elemente in dieser Reihenfolge in einen AVL-Baum eingefügt werden? Welcher optimale Suchbaum entsteht, wenn die Elemente die Zugriffshäufigkeiten $h(3) = 0.3$, $h(5) = 0.1$, $h(9) = 0.03$, $h(14) = 0.12$, $h(15) = 0.05$, $h(26) = 0.25$, $h(35) = 0.15$ haben?
- Hat ein AVL-Baum stets eine kleinere oder eine größere mittlere Suchdauer als ein optimaler Suchbaum?
- Hat ein AVL-Baum stets eine kleinere oder eine größere maximale Suchdauer als ein optimaler Suchbaum?

**Bestimmung der Inversionszahl:** Entwerfen Sie einen Algorithmus, der die Inversionszahl eines Arrays bestimmt. Welchen Aufwand in $O$-Notation hat Ihr Algorithmus?

**Programmieraufgabe zu Quicksort und Insertionsort:** Wir haben gelernt, dass Insertionsort bei Feldern mit geringem Fehlstand meist schneller als andere Verfahren ist. Ebenso haben wir gesehen, dass das Rekursionsende bei Quicksort zu vielen rekursiven Aufrufen mit sehr kleinen Feldern führt. Untersuchen Sie experimentell folgende Quicksort-Variante: Hat das Teilfeld höchstens noch 5 Elemente, so lassen wir es unsortiert. Ist der so modifizierte Algorithmus beendet, so wissen wir, dass Elemente nur innerhalb dieser kleinen Teilfelder unsortiert sein können. Je zwei Elemente aus verschiedenen Teilfeldern müssen bereits in der richtigen Reihenfolge stehen (warum)? Schätzen Sie ab, wie groß der Fehlstand nach diesem groben Quicksort-Verfahren maximal noch sein kann. Führt man nach dem groben Quicksort-Algorithmus noch Insertionsort aus, wird der geringe noch vorhandene Fehlstand garantiert mit geringem Aufwand korrigiert. Vergleichen Sie für zufällig gefüllte Arrays den normalen Quicksort-Algorithmus mit der hier beschriebenen groben Variante mit nachgeschaltetem Insertionsort. Welches Verfahren ist in der Praxis schneller?

# D Alte Klausuraufgaben

Lösungshinweise finden Sie unter `http://www.inf4dhbw.de.vu`

**Programmierung:** Gegeben sei der Datentyp für einen Binärbaum

```
struct BinBaum {
  int inhalt;
  BinBaum *l,*r;
};
```

Die Höhe eines Blattes ist bestimmt durch die Anzahl der Knoten auf dem Weg vom Blatt zur Wurzel. Wir wollen hier nun eine C++-Funktion

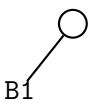
```
int minho(BinBaum * root)
```

schreiben, um in einem Binärbaum die minimale Höhe eines Blattes zu bestimmen. Der Parameter `root` zeigt dabei auf die Wurzel des Baums.

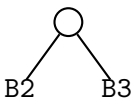
Beispiele: Der linke Baum hat eine minimale Höhe von 2 (das Blatt im rechten Unterbaum der Wurzel), der rechte Baum hat eine minimale Höhe von 4 (das einzige Blatt im Binärbaum)!

- Seien die Werte `m1=minho(B1)`, `m2=minho(B2)` und `m3=minho(B3)` für die drei Binärbäume `B1`, `B2` und `B3` bereits bestimmt. Geben Sie an, welchen Wert die Funktion `minho()` in folgenden 3 Fällen errechnet:

die minimale Höhe beträgt _____




die minimale Höhe beträgt _____




die minimale Höhe beträgt _____

- Schreiben Sie nun (nur) die Funktion `minho()`. Funktionen zur Erzeugung von Binärbäumen oder `int main()` sind nicht erforderlich.

```
int minho(BinBaum * root) {
  ...
}
```

**Programmierung:** Pythagoräische Zahlentripel `(a,b,c)` haben die Eigenschaft, dass `a*a+b*b=c*c` gilt, Beispiele: `(3,4,5)`, `(5,12,13)`, `(6,8,10)` sind Pythagoräische Zahlentripel, `(5,3,4)` und `(1,2,3)` sind keine.

Schreiben Sie eine C++-Funktion `void pyth(const int n)`, die alle Pythagoräischen Zahlentripel `(a,b,c)` mit der Eigenschaft `a*a+b*b=c*c` und `1<a<b<c<n` berechnet und ausgibt. Die Zahlentripel sollen in aufsteigender Reihenfolge bzgl. des Wertes `c` angegeben werden. Für `n=14` soll die Ausgabe somit

```
(3,4,5)
(6,8,10)
(5,12,13)
```

lauten. Achten Sie auch darauf, dass kein Zahlentripel doppelt ausgegeben wird. (Hinweis: es ist möglich die Aufgabe ohne Verwendung eines Sortieralgorithmus zu lösen! Funktionen zum Einlesen des Wertes `n` oder `int main()` sind nicht erforderlich.)

Beschreiben Sie anschließend, wie Sie Ihr Programm ändern müssten, wenn die Zahlentripel in aufsteigender Reihenfolge bzgl. des Wertes `a` angegeben werden sollen.

**Ein Baumalgorithmus:** Gegeben sei der Datentyp für einen Binärbaum

```
struct BinBaum {
  int inhalt;
  BinBaum *l,*r;
};
```

Die Knoten eines Binärbaumes kann man unterscheiden in „echte“ Blätter (Knoten ohne Kinder), „innere Knoten“ (Knoten mit genau 2 Kindern) und den Rest (Knoten mit genau einem Kind). Sie sollen hier eine C++-Funktion

```
int anzahl(BinBaum * root)
```

schreiben, die in einem Binärbaum die Anzahl dieser restlichen Knoten bestimmt (also die Anzahl der Knoten, die weder echte Blätter noch innere Knoten sind). Der Parameter `root` zeigt dabei auf die Wurzel des Baums.

Beispiele: Die gesuchte Anzahl beträgt im linken Baum 0 (kein Knoten hat genau 1 Kind), die Anzahl im rechten Baum ist 3 (alle Knoten bis auf das Blatt)!

- Seien die Werte `a1=anzahl(B1)`, `a2=anzahl(B2)` und `a3=anzahl(B3)` für die drei Binärbäume `B1`, `B2` und `B3` bereits bestimmt. Geben Sie an, welchen Wert die Funktion `anzahl()` in folgenden 3 Fällen errechnet:


- Schreiben Sie nun (nur) die Funktion `anzahl()`. Funktionen zur Erzeugung von Binärbäumen oder `int main()` sind nicht erforderlich.

```
int anzahl(BinBaum * root) {
  ...
}
```

**Binäre Suchbäume:**

- Fügen Sie die Zahlen 18, 12, 5, 13, 21, 42 und 37 in einen anfangs leeren binären Suchbaum ein. Es reicht die Angabe des Suchbaums, Sie müssen nicht das Verfahren des Einfügens beschreiben.
- Geben Sie eine andere Reihenfolge der Zahlen an, so dass beim Einfügen der Zahlen in dieser Reihenfolge derselbe binäre Suchbaum wie in der ersten Teilaufgabe entsteht, oder begründen Sie, warum es keine weitere solche Reihenfolge der Zahlen geben kann.
- Geben Sie eine andere Reihenfolge der Zahlen an, so dass beim Einfügen der Zahlen in dieser Reihenfolge der resultierende binäre Suchbaum minimale Höhe hat.

**Sortieren:**

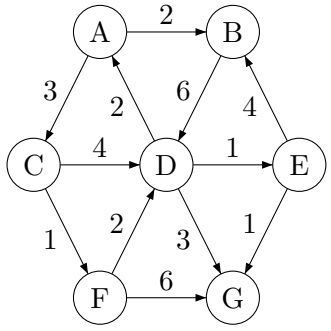
- Welcher Sortieralgorithmus wurde verwendet (falls es mehrere korrekte Antworten gibt, so genügt die Angabe von jeweils einem Algorithmus)? Begründungen sind hier nicht gefordert.
  - Gegeben ist das Feld mit den Werten
    (14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (14,21,52,18,63,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (14,21,52,18,42,63,37).
  - Gegeben ist das Feld mit den Werten
    (14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (14,21,52,18,63,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (14,21,18,52,63,42,37).
  - Zusatzaufgabe (2+1 Punkte): Gegeben ist das Feld mit den Werten
    (14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (14,63,52,21,18,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
    (63,14,52,21,18,42,37).
    Wie lautet der Inhalt nach dem dritten Tausch?
- Führen Sie mit dem Quicksort-Algorithmus den ersten Quicksort-Schritt (also ein komplettes Aufteilen in Elemente kleiner gleich und größer gleich dem Pivot-Element) in dem Feld mit den folgenden Zahlen durch. Wählen Sie als Pivot-Element das mittlere Element (42) und geben Sie an, welche beiden rekursiven Aufrufe nach diesem Quicksort-Schritt aufgerufen werden.

  Aus Ihrer Bearbeitung muss hervorgehen, welche Zahlen bei dem Prozess vertauscht wurden und wie diese zu vertauschenden Zahlen gefunden wurden. Ebenso muss ersichtlich sein, wie Sie die Parameter der beiden rekursiven Aufrufe ermittelt wurden.

  14 21 52 63 18 42 37 42 86 42 13

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:** Beschreiben Sie in 3-4 Sätzen das Vorgehen beim Dijkstra-Algorithmus (Sie müssen dabei nicht begründen, warum das Vorgehen korrekt ist).

Führen Sie nun den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an. Geben Sie die berechnete Entfernung d() für alle Knoten an (die Vorgänger-Knoten pred() müssen nicht mit berechnet werden).



**Analyse eines Sortieralgorithmus:** Sie haben in einem Buch folgenden Sortieralgorithmus gefunden:

```
const int N = 7;
int tosort[N] = {5,8,2,1,4,9,6};

void UnknownSort() {
  int z,h;
  for(int i=1;i<N;i++) {
    z=0;
    for(int j=0;j<i;j++) { if (tosort[j]<tosort[i]) z=z+1; } // zähle "<tosort[i]"
    h=tosort[i];
    for(int k=i;k>z;k--) { tosort[k]=tosort[k-1]; } // verschiebe
    tosort[z]=h;
  }
}
```

Dieser sortiert – so steht es im Buch – das Feld `tosort[]`. Die Aussage soll überprüft und die Laufzeit analysiert werden.

- Führen Sie den Algorithmus aus und geben Sie den Inhalt des Feldes `tosort[]` nach jedem Durchlauf der äußeren Schleife an. Beachten Sie dabei, dass in C/C++ Arrays mit Index 0 beginnen.
- Formulieren Sie die Idee, die hinter dem Algorithmus steckt, und begründen Sie, warum der Algorithmus tatsächlich sortiert.
- Geben Sie die Laufzeit des Algorithmus in $O$-Notation an.

**Listen-Programmierung:** Wir betrachten in dieser Aufgabe einfach verkettete Listen aus `int`-Zahlen, die sortiert sind. Der Anker einer Liste ist ein Zeiger auf das erste Element der Liste (oder 0, falls die Liste leer ist). Ihre Aufgabe ist, eine Funktion zu schreiben, die zwei Anker auf zwei sortierte einfach verkettete Listen erhält und den Anker auf eine neue Liste als Ergebnis zurückliefert. Dabei sollen die beiden übergebenen Listen erhalten bleiben! D. h., sie müssen die Elemente für die Ergebnisliste neu (mittels `new`) erzeugen. (Hinweis: eine Möglichkeit zur Lösung der Aufgabe ist, den Mischprozess vom Mergesort-Algorithmus, der dort auf Arrays durchgeführt wird, auf Listen zu übertragen.)

- Geben Sie eine C/C++-Datenstruktur `Liste` für eine einfach verkettete Liste von `int`-Werten an.
- Schreiben Sie eine Funktion `Liste* mische(Liste *eins, Liste *zwei)`, die als Parameter die Anker `eins` und `zwei` auf die erste bzw. zweite Liste erhält und als Ergebnis einen Anker auf eine Liste zurückgibt, die alle Elemente der ersten und zweiten Liste in sortierter Reihenfolge enthält. Beachten Sie dabei, dass die Listen jeweils leer sein können.
- Zusatzaufgabe: Geben Sie die Laufzeit Ihres Programms in $O$-Notation in Abhängigkeit der Größen der beiden Listen an. Die Liste `eins` soll dabei $n_1$ Elemente und die Liste `zwei` $n_2$ Elemente enthalten.

Hinweis: es ist nur C/C++-Code für die Datenstruktur und die Mische-Funktion gefordert. Sie müssen sich nicht um die Ein- und Ausgabe der Listen kümmern und auch kein Hauptprogramm (`int main()`) schreiben.

**Binäre Suchbäume:** Die Zahlen $x$, 7, 9, 5, 21, 42 und $y$ sollen in dieser Reihenfolge in einen anfangs leeren binären Suchbaum eingefügt werden. Die Werte $x$ und $y$ sind dabei Platzhalter mit folgenden Eigenschaften: der resultierende binäre Suchbaum hat minimale Höhe und alle Werte im Suchbaum sind verschieden.

- Geben Sie an, welche Werte die Variablen $x$ und $y$ unter Annahme dieser Eigenschaften haben können.
- Wählen Sie mögliche Werte für $x$ und $y$, fügen Sie die Zahlen in obiger Reihenfolge in einen anfangs leeren binären Suchbaum ein. Es reicht die Angabe des Suchbaums, Sie müssen nicht das Verfahren des Einfügens beschreiben. (Falls Sie die erste Teilaufgabe nicht lösen konnten, wählen Sie $x$=8 und $y$=13; der resultierende Suchbaum muss in diesem Fall nicht minimale Höhe haben.)
- Geben Sie die Zahlen des so entstandenen binären Suchbaums in Preorder, Inorder und Postorder an.

**Sortieren:**

- Führen Sie den Mergesort-Algorithmus mit folgenden Zahlen durch.

  52 18 42 37 24 86 13 23

  Machen Sie in der Bearbeitung deutlich, in welcher Reihenfolge die rekursiven Aufrufe beim Sortieren stattfinden (z. B., indem Sie den jeweils nächsten rekursiven Aufruf und jeweils das Ergebnis des Mischens zweier sortierter Teilfelder in eine eigene Zeile schreiben – oder nummerieren Sie die einzelnen Schritte soweit nötig durch).

- Führen Sie den Quicksort-Algorithmus mit folgenden Zahlen durch.

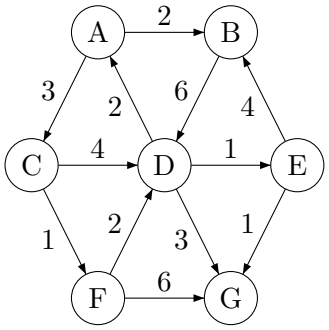
  52 18 42 37 24 86 13 23

  Es reicht, wenn Sie dabei die Quicksortschritte (Aufteilen des (Teil-)Felds in Elemente kleiner gleich und größer gleich dem Pivot-Element) der ersten 4 rekursiven Aufrufe durchführen (der erste rekursive Aufruf ist dabei `Quicksort(0,7)`). Wählen Sie als Pivot-Element dabei stets das mittlere des zu sortierenden Teilfeldes.

  Aus Ihrer Bearbeitung muss hervorgehen, welche Zahlen bei dem Prozess vertauscht wurden und wie diese zu vertauschenden Zahlen gefunden wurden. Ebenso muss ersichtlich sein, wie die Parameter der rekursiven Aufrufe ermittelt wurden.

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:** Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:

Zu Beginn ist lediglich die Entfernung zum Startknoten F mit d(F)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an. Geben Sie die berechnete Entfernung d() für alle Knoten an (die Vorgänger-Knoten pred() müssen nicht mit berechnet werden).
Ist die Abarbeitungsreihenfolge der Knoten eindeutig? Falls ja, begründen Sie dies; falls nein, geben Sie an, an welchen Stellen Sie auch einen anderen Knoten aus der Menge R hätten wählen können.




**Analyse eines Listenalgorithmus:** Sie haben in einem Buch folgenden Listenalgorithmus gefunden:

```
struct Liste{
  int value;
  Liste* next;
};

Liste* miste(Liste *eins, Liste *zwei){
  if (eins==zwei) return eins;
  if (eins==0) return zwei; // Fall beide 0 oder eins 0 abgehakt
  if (zwei==0) return eins; // Fall zwei 0 auch abgehakt
  if (eins->value<zwei->value) {
    eins->next = miste(eins->next,zwei);
    return eins;
  } else {
    zwei->next = miste(eins,zwei->next);
    return zwei;
  }
}

void putliste(Liste* p){
  while (p) { cout << p->value << ' '; p=p->next; }
  cout << endl;
}
```

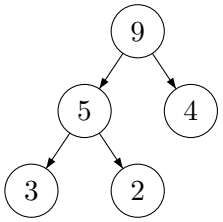
Das Buch verrät jedoch nicht, was der Algorithmus `miste` tut.

- Gegeben sind die Zeiger `p1` auf die Liste mit den Elementen (5,12,21,34) und `p2` auf die Liste mit den Elementen (17,42,47,53). Stellen Sie die Listen graphisch dar (jedes Element einer Liste als Kästchen mit dem `next`-Zeiger als Pfeil auf das entsprechende nächste Kästchen) und führen Sie den Algorithmus `miste` mit den Parametern `p1` und `p2` durch (Aufruf `p3=miste(p1,p2);`). Welche Zeiger werden verändert, welche Elemente neu erzeugt? Welche Ausgabe ergibt nach diesem Aufruf die Befehlsfolge `putliste(p1); putliste(p2); putliste(p3);`?

- Geben Sie die Laufzeit des Algorithmus in $O$-Notation abhängig von der Anzahl $n_1$ der Elemente der ersten Liste und der Anzahl $n_2$ der Elemente der zweiten Liste an.
- Zusatzaufgabe (knifflig): was kann passieren, wenn die Abfrage `eins==zwei` entfällt?

**Programmierung:** Wir schreiben in dieser Aufgabe eine Ausgaberoutine für Binärbäume, deren Knoten mit natürlichen Zahlen beschriftet sind.

Ziel ist, für einen Binärbaum eine Ausgabe der Form a+b+c+d+...+x+y zu erhalten. Die Ausgabe für nebenstehenden Baum soll also z. B. 3+5+2+9+4 lauten (eine Lösungsmöglichkeit führt auf diese Reihenfolge der Zahlen, Ihre Lösung muss die Zahlen nicht unbedingt in dieser Reihenfolge auflisten – eine weitere naheliegende Lösung führt auf die Ausgabe 9+5+3+2+4)




Die geforderte Ausgabe lässt sich leicht rekursiv erzeugen: Ist der Baum leer, so ist nichts auszugeben, besteht der Baum nur aus einem Knoten, so ist nur dieser auszugeben. Überlegen Sie sich, welche Ausgabe nötig ist,

- wenn beide Unterbäume nicht leer sind,
- wenn genau ein Unterbaum nicht leer ist.

- Geben Sie eine C/C++-Datenstruktur `BinBaum` für einen Binärbaum mit "`unsigned int`"-Werten an. Schreiben Sie eine (rekursive) Funktion `ausgabe`, die einen Zeiger auf die Wurzel eines Binärbaums übergeben bekommt und die Zahlen im Binärbaum durch Plus-Zeichen getrennt ausgibt. Nach der letzten Zahl darf kein Pluszeichen folgen!
- Zusatzaufgabe: Schreiben Sie Ihre Funktion so, dass die Ausgabe die Form a+b+c+d+...+x+y=z hat. In obigen Beispiel soll die Ausgabe also 3+5+2+9+4=23 lauten. (Hinweis: die Berechnung der Summe der Zahlen lässt sich leicht in die Funktion integrieren. Die Ausgabe des Gleichheitszeichens und der Summe kann außerhalb der Funktion mit einem weiteren Befehl erfolgen.)
- Zusatzaufgabe: Geben Sie die Laufzeit Ihres Programms in $O$-Notation in Abhängigkeit der Anzahl `n` der Elemente im Binärbaum an.

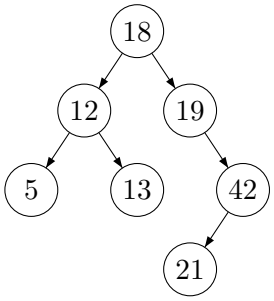
Hinweis: es ist nur C/C++-Code für die Datenstruktur und die `ausgabe`-Funktion gefordert. Sie müssen sich nicht um die Eingabe und das Einfügen der Werte im Binärbaum kümmern und auch kein Hauptprogramm (`int main()`) schreiben.

**Binäre Suche, Interpolationssuche:**
Gegeben ist ein Array mit den Werten (5,6,8,10,25,28,29). Es soll der Wert 25 gesucht werden.

- Geben Sie an, mit welchen Elementen bei **binärer Suche** die 25 verglichen wird, bis sie gefunden wird.

- Geben Sie an, mit welchen Elementen bei **Interpolationssuche** die 25 verglichen wird, bis sie gefunden wird.
- Welche Eigenschaft muss ein Array mit Zahlen haben, damit man in diesem binäre Suche und Interpolationssuche anwenden kann? (Kurze Antwort mit einem Satz Begründung reicht)

**Löschen in binären Suchbäumen:** Gegeben ist folgender binäre Suchbaum:




- Geben Sie an, welcher Suchbaum entsteht, wenn die 18 gelöscht wird.
- Beschreiben Sie in 3 Sätzen, welche Fälle beim Löschen in binären Suchbäumen entstehen können und wie diese behandelt werden.

**AVL-Bäume:** Fügen Sie in einen anfangs leeren AVL-Baum folgende Zahlen in dieser Reihenfolge ein: 9, 7, 2, 6, 3, 1, 4. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

**Insertionsort:** Gegeben ist ein Array mit den Zahlen (9,7,2,6,3,1,4).

Führen Sie Insertionsort durch:

- Geben Sie den Inhalt des Arrays nach jedem kompletten Durchlauf der äußeren Schleife an.
- Tragen Sie jeweils die Anzahl der in diesem Durchlauf durchgeführten Vertauschungen ein.

Hinweis: die Anzahl der Zeilen in der Tabelle ist größer als die tatsächlich auszufüllende Anzahl der Zeilen.

| Durchlauf | | | | | | | | Vertauschungen |
|---|---|---|---|---|---|---|---|---|
| vor 1. | 9 | 7 | 2 | 6 | 3 | 1 | 4 | - |
| nach 1. | | | | | | | | |
| nach 2. | | | | | | | | |
| nach 3. | | | | | | | | |
| nach 4. | | | | | | | | |
| nach 5. | | | | | | | | |
| nach 6. | | | | | | | | |
| nach 7. | | | | | | | | |
| nach 8. | | | | | | | | |
| nach 9. | | | | | | | | |
| nach 10. | | | | | | | | |
| nach 11. | | | | | | | | |
| nach 12. | | | | | | | | |
| nach 13. | | | | | | | | |
| nach 14. | | | | | | | | |
| nach 15. | | | | | | | | |

Wie groß ist der Fehlstand (Inversionszahl) des obigen Feldes (9,7,2,6,3,1,4)?

**Quicksort:** Bei Quicksort wird als Pivot-Element das Element in der Mitte gewählt (Index `(l+r)/2`, hier Index 10). Folgendes Feld sei das Ergebnis der ersten Iteration bei Quicksort (also Wandern der Zeiger mit Vertauschungen bis die beiden Zeiger übereinander hinweg gelaufen sind):

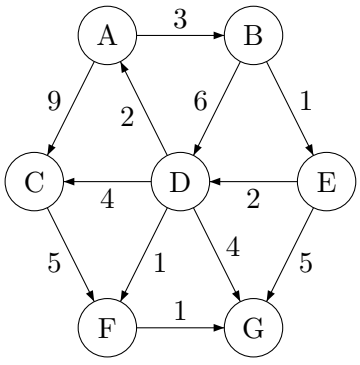
10 22 15 14 16 6 25 9 31 56 91 75 99 88 94 55 79 98 73 77 59

Geben Sie eine Möglichkeit an, welches Pivot-Element gewählt wurde, und begründen Sie Ihre Antwort kurz.

**Zusatzaufgabe:** Ist das von Ihnen gefundene Element das einzig mögliche?

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.



| | Menge R | Ergebnis von delete_min(R) | berechnete Entfernungen A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|
| 0. | $\emptyset$ | „Initialisierung“ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1. | $\{A\}$ | | | | | | | | |
| 2. | | | | | | | | | |
| 3. | | | | | | | | | |
| 4. | | | | | | | | | |
| 5. | | | | | | | | | |
| 6. | | | | | | | | | |
| 7. | | | | | | | | | |
| 8. | | | | | | | | | |

**Ein Baumalgorithmus:** In dieser Aufgabe werden einige grundlegende Eigenschaften von binären Suchbäumen untersucht und in ein Programm umgesetzt. (Hinweis: die Fragen weisen einen möglichen Weg, die C/C++-Funktion zu realisieren – sie können die Programmieraufgabe aber auch unabhängig von Ihren vorherigen Antworten lösen.)

Es reichen jeweils kurze Antworten mit je einem Satz Begründung.

- Wo befindet sich in einem binären Suchbaum das größte Element? Warum kann der Knoten mit dem größten Wert in einem binären Suchbaum kein rechtes Kind haben?
- Wo liegt das zweitgrößte Element, wenn das größte Element ein linkes Kind hat? Wo liegt das zweitgrößte Element, wenn das größte Element ein Blatt ist?
- Geben Sie eine C/C++-Datenstruktur für einen binären Suchbaum an. Der Datentyp für eine Variable, die auf die Wurzel eines binären Suchbaums zeigt, soll dabei `binSBaumPtr` heißen.
- Schreiben Sie eine C/C++-Funktion `void zweites(const binSBaumPtr root)`, die als Parameter den Zeiger auf die Wurzel eines binären Suchbaums bekommt. Die geforderte Ausgabe ist:
  - Ist der Baum leer, so soll die Meldung „`Baum ist leer.`“ ausgegeben werden.
  - Hat der Baum nur einen Knoten, so soll die Meldung „`Baum hat nur einen Knoten.`“ ausgegeben werden.
  - Hat der Baum mindestens zwei Knoten, so soll die Meldung „`Das zweitgrößte Element im Baum ist die` $x$“ ausgegeben werden, wobei $x$ durch den entsprechenden Wert zu ersetzen ist.

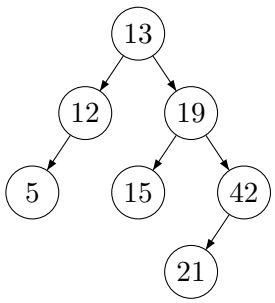
  Zu dem Baum aus Aufgabe 2 soll die Ausgabe also „`Das zweitgrößte Element im Baum ist die 21`“ lauten.
- Geben Sie den Aufwand Ihres Algorithmus (für den schlechtesten Fall) in $O$-Notation an.
- Wieviel Schritte benötigt Ihr Algorithmus im besten Fall? Skizzieren Sie einen Baum mit 10 Knoten, der mit Ihrem Algorithmus möglichst schnell bearbeitet wird.

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe des Baumes kümmern.

**Binäre Suche, Interpolationssuche:**
Gegeben ist ein Array mit den Werten (5,6,9,10,25,28,29). Es soll der Wert 9 gesucht werden.

- Geben Sie an, mit welchen Elementen bei **binärer Suche** die 9 verglichen wird, bis sie gefunden wird.
- Geben Sie an, mit welchen Elementen bei **Interpolationssuche** die 9 verglichen wird, bis sie gefunden wird.
- Sind diese Aussagen richtig oder falsch? (geben Sie jeweils ein passendes Gegenbeispiel oder eine kurze Begründung an)
  - Binäre Suche benötigt immer höchstens so viele Vergleiche wie Interpolationssuche.
  - Binäre Suche benötigt immer mindestens so viele Vergleiche wie Interpolationssuche.

**Löschen in binären Suchbäumen:** Gegeben ist folgender binäre Suchbaum:




- Werden in einem binären Suchbaum Knoten mit zwei Kindern gelöscht, so werden diese durch das nächstgrößere (Inordernachfolger) oder nächstkleinere Element (Inordervorgänger) ersetzt. Geben Sie die beiden Suchbäume an, die durch Löschen des Elements 13 entstehen können.

  Was muss man nach dem Ersetzen des zu löschenden Elements noch tun?
- Beschreiben Sie in 2 Sätzen, wie der Inordervorgänger oder Inordernachfolger gefunden werden kann.

**AVL-Bäume:** Fügen Sie in einen anfangs leeren AVL-Baum folgende Zahlen in dieser Reihenfolge ein: 1, 3, 8, 4, 7, 9, 6. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

**Insertionsort:** Gegeben ist ein Array mit den Zahlen (9,7,3,4,2,6,5).

Führen Sie Insertionsort durch:

- Geben Sie den Inhalt des Arrays nach jedem kompletten Durchlauf der äußeren Schleife an.

- Tragen Sie jeweils die Anzahl der in diesem Durchlauf durchgeführten Vertauschungen ein.

Hinweis: die Anzahl der Zeilen in der Tabelle ist größer als die tatsächlich auszufüllende Anzahl der Zeilen.

| Durchlauf | | | | | | | | Vertauschungen |
|---|---|---|---|---|---|---|---|---|
| vor 1. | 9 | 7 | 3 | 4 | 2 | 6 | 5 | - |
| nach 1. | | | | | | | | |
| nach 2. | | | | | | | | |
| nach 3. | | | | | | | | |
| nach 4. | | | | | | | | |
| nach 5. | | | | | | | | |
| nach 6. | | | | | | | | |
| nach 7. | | | | | | | | |
| nach 8. | | | | | | | | |
| nach 9. | | | | | | | | |
| nach 10. | | | | | | | | |
| nach 11. | | | | | | | | |
| nach 12. | | | | | | | | |
| nach 13. | | | | | | | | |
| nach 14. | | | | | | | | |
| nach 15. | | | | | | | | |

Wie groß ist der Fehlstand (Inversionszahl) des obigen Feldes (9,7,3,4,2,6,5)?

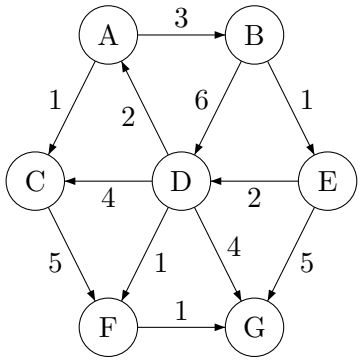
**Bei Quicksort** wird als Pivot-Element das Element in der Mitte gewählt (Index `(l+r)/2`). Folgendes Feld sei gegeben. Führen Sie die erste Iteration bei Quicksort durch (also Wandern der Zeiger mit Vertauschungen bis die beiden Zeiger übereinander hinweg gelaufen sind) – in Ihrer Bearbeitung muss ersichtlich sein, welche Elemente getauscht wurden und nach welchem Kriterium das Wandern der Zeiger beendet wird:

10 72 15 14 66 6 45 49 31 56 91 75 39 88 94 55 19 18 73

Geben Sie an, mit welchen Indexgrenzen die beiden resultierenden rekursiven Aufrufe von Quicksort erfolgen.

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten B mit d(B)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.



| | Menge R | Ergebnis von delete_min(R) | berechnete Entfernungen A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|
| 0. | $\emptyset$ | „Initialisierung“ | $\infty$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1. | {B} | | | | | | | | |
| 2. | | | | | | | | | |
| 3. | | | | | | | | | |
| 4. | | | | | | | | | |
| 5. | | | | | | | | | |
| 6. | | | | | | | | | |
| 7. | | | | | | | | | |
| 8. | | | | | | | | | |

**Ein Baumalgorithmus:** Bei Bäumen spielt oft das Level (Ebene) eines Knotens eine Rolle – der Wurzelknoten ist auf Level 1, dessen Kinder sind auf Level 2, usw. In dieser Aufgabe sollen die Werte in den Knoten jeweils mit deren Level gewichtet aufsummiert werden. Für den Baum auf Seite 104 soll der Wert

$$1 \cdot 13 + 2 \cdot 12 + 3 \cdot 5 + 2 \cdot 19 + 3 \cdot 15 + 3 \cdot 42 + 4 \cdot 21 \quad = \quad 345$$

berechnet werden.

- Geben Sie eine C/C++-Datenstruktur für einen binären Baum an. Die Werte im Baum sollen vom Datentyp `int` sein. Der Datentyp für eine Variable, die auf die Wurzel eines binären Baums zeigt, soll dabei `binBaumPtr` heißen.
- Schreiben Sie eine C/C++-Funktion `int gesumme(const binBaumPtr root)`, die als Parameter den Zeiger auf die Wurzel eines binären Baums bekommt und als Ergebnis (mit dem Befehl `return`) die gewichtete Summe zurückgibt (eine Ausgabe per `printf` oder `cout` ist nicht gefordert). Ist der Baum leer, so ist der Rückgabewert 0.

  (Hinweis: wahrscheinlich werden Sie eine weitere Hilfsfunktion benötigen)
- Geben Sie den Aufwand für den schlechtesten Fall in $O$-Notation an.
- Wieviel Schritte benötigen Sie im besten Fall?

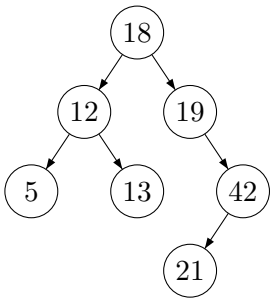
Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe des Baumes kümmern.

**Binäre Suche, Interpolationssuche:**
Gegeben ist ein Array mit den Werten (5,6,8,10,23,26,29).

- Geben Sie an, welche Elemente bei Verwendung der **binären Suche** beim ersten Vergleich gefunden werden können.
- Geben Sie an, welche Elemente bei Verwendung der **Interpolationssuche** beim ersten Vergleich gefunden werden können.

  Geben Sie jeweils eine kurze Begründung an (1–2 Sätze genügen).

**Binäre Suchbäume:** Gegeben ist folgender binäre Suchbaum:




- Geben Sie die Pre-, In- und Postorderreihenfolge des links stehenden binären Suchbaums an.
- Wie verändert sich die Preorderreihenfolge, wenn Sie ein weiteres Element in den binären Suchbaum einfügen?

**AVL-Bäume:** Fügen Sie in einen anfangs leeren AVL-Baum folgende Zahlen in dieser Reihenfolge ein: 3, 8, 9, 7, 4, 1, 6. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

**Ein Algorithmus:** Gegeben ist folgendes C++-Programmfragment:

```
const int n=7;
int a[n]={9,7,2,6,3,1,4};
int i,j;
int c=0;
for(i=1;i<n,i++) {
  for(j=0;j<i;j++) {
    if (a[j]>a[i]) {
      c=c+1;
    }
  }
}
```

- Welchen konkreten Wert hat die Variable `c` nach Ausführung des obigen Programmfragments?
- Welchen Wert hat die Variable in Abhängigkeit der Belegung des Feldes `a[]`? Begründen Sie Ihre Antwort.
- Wie groß ist der Aufwand des Programmfragments in Abhängigkeit von `n` in $O$-Notation?

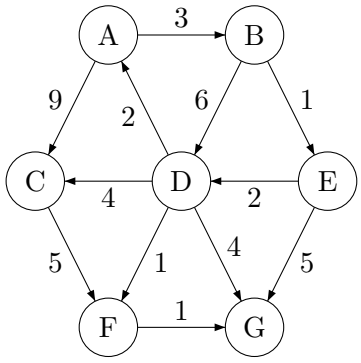
**Bei Quicksort** wird als Pivot-Element das Element in der Mitte gewählt (Index `(l+r)/2`, hier das 11-te der 21 Elemente). Folgendes Feld soll mit Quicksort sortiert werden. Führen Sie die erste Iteration bei Quicksort durch (also das Wandern der Zeiger mit Vertauschungen bis die beiden Zeiger übereinander hinweg gelaufen sind):

70 22 65 44 16 56 25 9 31 46 49 75 29 88 94 15 19 28 23 77 39

Geben Sie an, welche Vertauschungen durchgeführt werden und welche beiden rekursiven Aufrufe am Ende erfolgen.

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten E mit d(E)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.




| | Menge R | Ergebnis von delete_min(R) | berechnete Entfernungen A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|
| 0. | $\emptyset$ | „Initialisierung“ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0 | $\infty$ | $\infty$ |
| 1. | $\{E\}$ | | | | | | | | |
| 2. | | | | | | | | | |
| 3. | | | | | | | | | |
| 4. | | | | | | | | | |
| 5. | | | | | | | | | |
| 6. | | | | | | | | | |
| 7. | | | | | | | | | |
| 8. | | | | | | | | | |

**Ein Baumalgorithmus:** In dieser Aufgabe werden wir ein einfaches Programm entwerfen, das zu einem binären Suchbaum und einem Parameter $k$ das $k$-te Element in diesem Suchbaum bestimmt, und in C/C++-Code umsetzen. (Hinweis: die Fragen weisen einen möglichen Weg, die C/C++-Funktion zu realisieren – sie können die Programmieraufgabe aber auch unabhängig von Ihren vorherigen Antworten lösen.)

Nehmen Sie an, Sie kennen die Anzahl $nl$ der Knoten im linken Unterbaum des Suchbaums.

- In welchem mathematischen Kleiner-Größer-Gleich-Verhältnis stehen $nl$ und $k$, wenn das $k$-te Element sich im linken Unterbaum befindet? Das wievielte Element ist es dann, wenn man nur den linken Unterbaum betrachtet?
- In welchem mathematischen Kleiner-Größer-Gleich-Verhältnis stehen $nl$ und $k$, wenn das $k$-te Element die Wurzel des Baumes ist?
- In welchem mathematischen Kleiner-Größer-Gleich-Verhältnis stehen $nl$ und $k$, wenn das $k$-te Element sich im rechten Unterbaum befindet? Das wievielte Element ist es dann im rechten Unterbaum?
- Geben Sie eine C/C++-Datenstruktur für einen binären Suchbaum an. Der Datentyp für eine Variable, die auf die Wurzel eines binären Suchbaums zeigt, soll dabei `bstPtr` heißen.
- Schreiben Sie eine C/C++-Funktion `void k_tes_Element(const bstPtr root, const int k)`, die als Parameter den Zeiger auf die Wurzel eines binären Suchbaums und eine Zahl $k$ bekommt. Die geforderte Ausgabe ist:

– Ist der Baum leer, so soll die Meldung „`Baum ist leer.`“ ausgegeben werden.
– Ist das eingegebene $k \leq 0$ oder größer als die Anzahl der Knoten im Baum, so soll die Meldung „`Parameter k ungültig!`“ ausgegeben werden.
– Ansonsten soll die Meldung „`Das k-te Element im Baum ist die x`“ ausgegeben werden, wobei `k` und `x` durch die entsprechenden Werte zu ersetzen sind.

Zu dem Baum auf Seite 107 soll mit Parameter `k=6` die Ausgabe also „`Das 6-te Element im Baum ist die 21`“ lauten.

- Skizzieren Sie den Fall, für den Ihr Algorithmus möglichst lange braucht und geben Sie die Laufzeit in $O$-Notation an.

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe des Baumes kümmern.

Hinweis: Sie werden zusätzlich zur Funktion vermutlich ein oder zwei separate rekursive Hilfsfunktionen schreiben müssen. Es lässt sich zwar auch alles in einer Funktion integrieren, dies ist aber deutlich schwieriger umzusetzen.

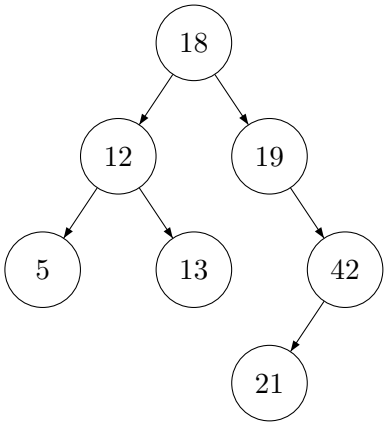
**Interpolationssuche:** Gegeben ist ein Array mit den Werten (5,8,$x$,15,20,23,29) mit einem unbekannten Wert $x$. Es soll der Wert $x$ gesucht werden.

- Geben Sie die Formel an, mit der bei der Interpolationssuche der nächste zu betrachtende Index berechnet wird, verwenden Sie dabei `su` für das zu suchende Element, `a[]` für das Array sowie `kl` und `gr` für den kleinsten bzw. größten Index, der im Array noch zu betrachten ist. (Begründung ist nicht erforderlich)

- Geben Sie an, welche Werte das Element $x$ im Array haben kann, damit dieses beim ersten Vergleich bereits gefunden wird (mit kurzer Begründung, 2 Sätze genügen)

**Ein Algorithmus auf Binärbäumen:**

Gegeben ist folgender Binärbaum:




und folgende Funktion:

```
void modify(BinBaumPtr root)
{
  if(root!=NULL)
  {
    modify(root->left);
    cout << root->value << ' '; // ein int-Wert
    BinBaumPtr swap=root->left;
    root->left=root->right;
    root->right=swap;
    modify(root->left);
  }
}
```

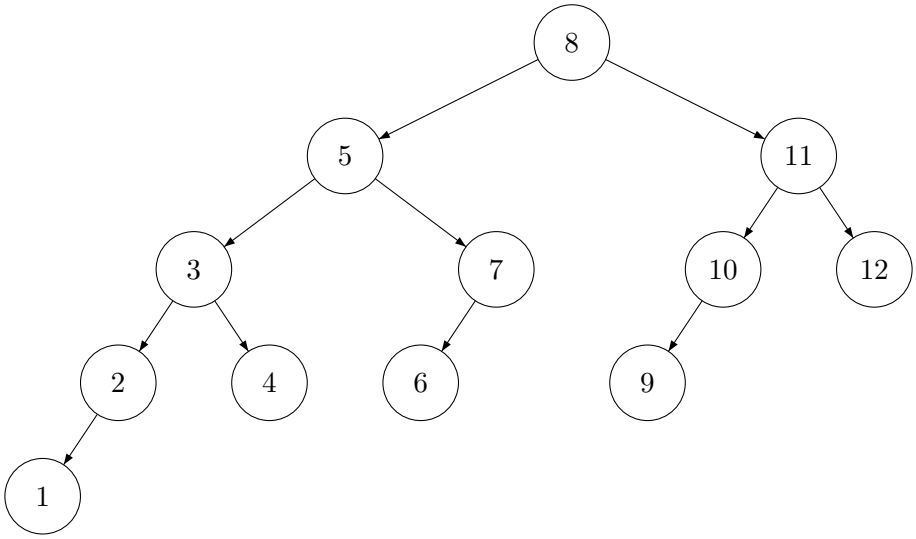
- Geben Sie die fehlende(n) C/C++-Datenstruktur(en) für die Funktion an, sodass diese vom Compiler übersetzt werden kann. Der Datentyp `BinBaumPtr` stehe dabei für einen Zeiger, der auf die Wurzel eines Binärbaums zeigt.

- Geben Sie an, welche Ausgabe (Zeile `cout`) der Algorithmus bei obigem Binärbaum liefert. Zeichnen Sie den Binärbaum, wie er durch den Ablauf der Funktion verändert wurde.
- Beschreiben Sie in 1-2 Sätzen, was die Funktion tut.
- Geben Sie den Aufwand der Funktion in $O$-Notation an.

**AVL-Bäume:** Fügen Sie in einen anfangs leeren AVL-Baum folgende Zahlen in dieser Reihenfolge ein: 1, 3, 5, 9, 7. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

Welche der Zahlen 0, 2, 4, 6 und 8 lassen sich in den Baum als sechstes Element einfügen, ohnedass eine Rotation dabei erfolgt?

**Löschen in AVL-Bäumen:** Gegeben ist folgender AVL-Baum:




Löschen Sie in diesem AVL-Baum die 12, geben Sie den Baum nach jeder dabei notwendigen Rotation an. Unterbäume, die sich dabei nicht ändern, brauchen nicht komplett neu gezeichnet zu werden.

**Heapsort:** Gegeben ist ein Array mit den Zahlen (9,7,4,6,3,1,2).

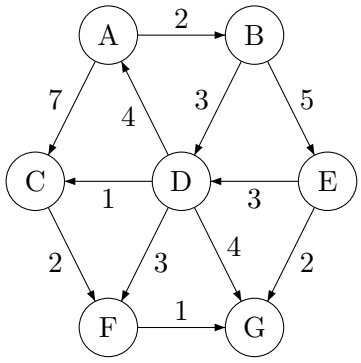
Bei Heapsort wird ein Max-Heap verwendet (größtes Element steht „oben“), der in der Phase des Heapaufbaus hergestellt wird.

Überprüfen Sie, ob das oben gegebene Array die Heapeigenschaft hat. Geben Sie an, welche konkreten Vergleiche dabei durchgeführt werden (z. B. $6 \geq 4$ oder ähnlich, eine allgemeine Beschreibung ist nicht gefordert).

Führen Sie nun den Heapsort-Algorithmus durch. Gegen Sie dabei jeweils den Heap an, nachdem das jeweils größte Element mit dem jeweils letzten Element des Heaps vertauscht wurde und die Heapeigenschaft durch Absinkenlassen der neuen Wurzel wiederhergestellt wurde. (Es reicht also die Angabe eines Heaps mit 6 Elementen, dann ein Heap mit 5 Elementen, analog mit 4, 3, 2 und 1 Element. Sie können dabei die Baum-Darstellung wählen.)

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. (9 Punkte) Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.

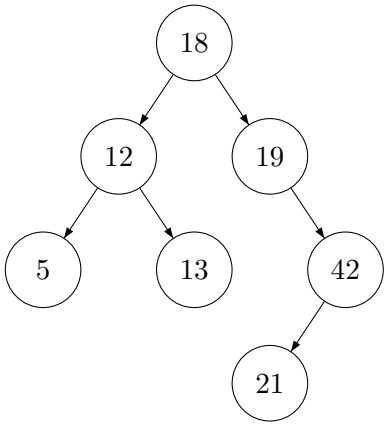



| | Menge R | Ergebnis von delete_min(R) | berechnete Entfernungen A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|
| 0. | $\emptyset$ | „Initialisierung“ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1. | {A} | | | | | | | | |
| 2. | | | | | | | | | |
| 3. | | | | | | | | | |
| 4. | | | | | | | | | |
| 5. | | | | | | | | | |
| 6. | | | | | | | | | |
| 7. | | | | | | | | | |
| 8. | | | | | | | | | |

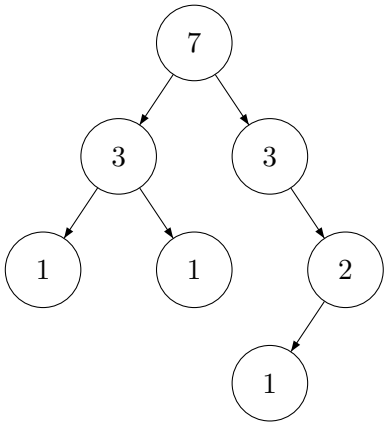
Geben Sie den Kürzeste-Wege-Baum an, also den Baum, der durch die während der Berechnung gemerkten Vorgänger (`pred[]`) bestimmt ist.

**Ein Suchbaumalgorithmus:** In dieser Aufgabe soll zunächst zu einem gegebenen binären Suchbaum ein strukturgleicher Binärbaum erstellt werden, sodass jede Knotenbeschriftung der Anzahl der Knoten im jeweiligen Unterbaum entspricht. Mithilfe dieses so erstellten Baumes kann nun das $k$-kleinste Element im Suchbaum schnell ermittelt werden.

Ein Beispiel: zu dem Baum auf Seite 109




wird zunächst der folgende Baum erstellt




Soll nun das 3-te Element im linken Suchbaum gefunden werden, so hat der linke Unterbaum bereits 3 Knoten, das gesuchte Element ist somit dort zu finden. Im Unterbaum mit Wurzel 12 sucht man also weiterhin das dritte Element; dieses findet man in dessen rechtem Unterbaum (der linke Unterbaum hat nur 1 Knoten, die Wurzel wäre der zweite Knoten). Es wird folglich

im Unterbaum mit der Wurzel 13 nun das erste Element gesucht. Da der linke Unterbaum der 13 leer ist, ist das gesuchte Element die 13.

- Geben Sie eine C/C++-Datenstruktur für einen binären Suchbaum oder Binärbaum an. Der Datentyp für eine Variable, die auf die Wurzel eines solchen Baums zeigt, soll dabei `bbPtr` heißen.
- Schreiben Sie eine C/C++-Funktion `bbPtr anzahlKopie(const bbPtr root)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, der kopiert werden soll, und als Ergebnis den Zeiger auf die Wurzel des erstellten Baumes zurückliefert. (Wenn Sie diese Aufgabe nicht komplett lösen können, schreiben Sie alternativ eine C/C++-Funktion, die in einem Binärbaum lediglich die Knotenbeschriftung so ändert, dass diese jeweils der Anzahl der Knoten im Unterbaum entspricht.)
- Schreiben Sie eine weitere C/C++-Funktion `int baumselect(const bbPtr root, const bbPtr anzKopie, const int k)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, in dem gesucht werden soll, einen weiteren Zeiger auf die Wurzel des Binärbaums, der durch die obige Funktion `anzahlKopie` entstanden ist, sowie eine Zahl `k`. Rückgabewert der Funktion soll das nach oben beschriebener Methode gefundene Element sein (falls `k` zu klein oder zu groß ist, so sei das Ergebnis `-1`).
- Geben Sie den Aufwand Ihres Algorithmus `anzahlKopie(...)` in $O$-Notation an.
- Mit welchem Aufwand im Mittel würden Sie für einen Aufruf Ihrer Funktion `baumselect(...)` rechnen?

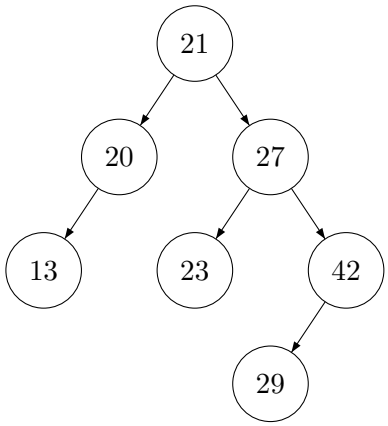
Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe eines Baumes kümmern.

**Zusatzaufgabe (schwer):** Man hätte die Knoten in der Kopie auch einfach in Inorderreihenfolge nummerieren können und somit einfach das `k` in der Kopie suchen können. Welchen Vorteil könnte das obige Vorgehen über die Anzahl der Knoten in den Unterbäumen haben?

**Binäre Suche, Interpolationssuche:** Gegeben ist ein Array mit den Werten (13,14,17,18,33,36,37). Es soll der Wert 17 gesucht werden.

- Geben Sie an, mit welchen Elementen bei **binärer Suche** die 17 verglichen wird, bis sie gefunden wird.
- Geben Sie an, mit welchen Elementen bei **Interpolationssuche** die 17 verglichen wird, bis sie gefunden wird.

**Löschen in binären Suchbäumen:** Gegeben ist folgender binäre Suchbaum:




- Werden in einem binären Suchbaum Knoten mit zwei Kindern gelöscht, so werden diese durch das nächstgrößere (Inordernachfolger) oder nächstkleinere Element (Inordervorgänger) ersetzt. Geben Sie die beiden Suchbäume an, die durch Löschen des Elements 21 entstehen können.

  Was muss man nach dem Ersetzen des zu löschenden Elements noch tun?

- Beschreiben Sie in 2 Sätzen, wie der Inordervorgänger oder Inordernachfolger gefunden werden kann.

**AVL-Bäume:** Fügen Sie in einen anfangs leeren AVL-Baum folgende Zahlen in dieser Reihenfolge ein: 2, 4, 8, 5, 7, 9, 6. Wenn eine Rotation notwendig ist, geben Sie den AVL-Baum jeweils vor und nach dieser Rotation an. Geben Sie in jedem Fall den AVL-Baum an, nachdem alle Elemente eingefügt wurden.

**Insertionsort:** Gegeben ist ein Array mit den Zahlen (8,6,2,3,1,5,4).

Führen Sie Insertionsort durch:

- Geben Sie den Inhalt des Arrays nach jedem kompletten Durchlauf der äußeren Schleife an.
- Tragen Sie jeweils die Anzahl der in diesem Durchlauf durchgeführten Vertauschungen ein.

Hinweis: die Anzahl der Zeilen in der Tabelle ist größer als die tatsächlich auszufüllende Anzahl der Zeilen.

| Durchlauf | | | | | | | | Vertauschungen |
|---|---|---|---|---|---|---|---|---|
| vor 1. | 8 | 6 | 2 | 3 | 1 | 5 | 4 | - |
| nach 1. | | | | | | | | |
| nach 2. | | | | | | | | |
| nach 3. | | | | | | | | |
| nach 4. | | | | | | | | |
| nach 5. | | | | | | | | |
| nach 6. | | | | | | | | |
| nach 7. | | | | | | | | |
| nach 8. | | | | | | | | |
| nach 9. | | | | | | | | |
| nach 10. | | | | | | | | |
| nach 11. | | | | | | | | |
| nach 12. | | | | | | | | |
| nach 13. | | | | | | | | |
| nach 14. | | | | | | | | |
| nach 15. | | | | | | | | |

Wie groß ist der Fehlstand (Inversionszahl) des obigen Feldes (8,6,2,3,1,5,4)?

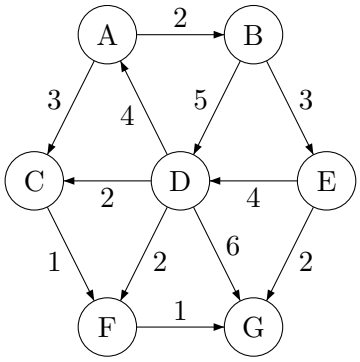
**Bei Quicksort** wird als Pivot-Element das Element in der Mitte gewählt (Index `(l+r)/2`). Folgendes Feld sei gegeben. Führen Sie die erste Iteration bei Quicksort durch (also Wandern der Zeiger mit Vertauschungen bis die beiden Zeiger übereinander hinweg gelaufen sind) – in Ihrer Bearbeitung muss ersichtlich sein, welche Elemente getauscht wurden und nach welchem Kriterium das Wandern der Zeiger beendet wird:

11 73 16 15 67 7 46 50 32 57 92 76 42 89 95 56 21 20 74

Geben Sie an, mit welchen Indexgrenzen die beiden resultierenden rekursiven Aufrufe von Quicksort erfolgen.

**Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

Führen Sie den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie in jeder Zeile den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an, den durch delete_min gewählten Knoten und die berechneten Entfernungen d() am Ende des Durchlaufs für alle Knoten.

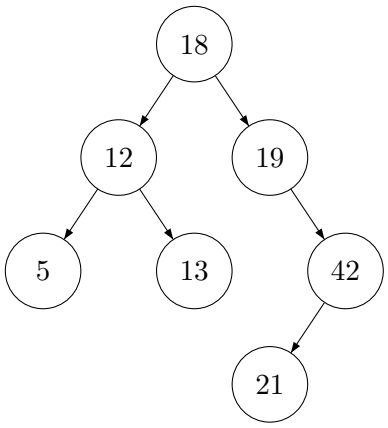


| | Menge R | Ergebnis von delete_min(R) | berechnete Entfernungen A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|
| 0. | $\emptyset$ | „Initialisierung“ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1. | {A} | | | | | | | | |
| 2. | | | | | | | | | |
| 3. | | | | | | | | | |
| 4. | | | | | | | | | |
| 5. | | | | | | | | | |
| 6. | | | | | | | | | |
| 7. | | | | | | | | | |
| 8. | | | | | | | | | |

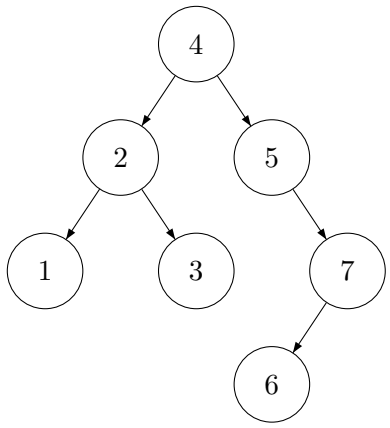
Geben Sie den Kürzeste-Wege-Baum an, also den Baum, der durch die während der Berechnung gemerkten Vorgänger (`pred[]`) bestimmt ist.

**Ein Suchbaumalgorithmus:** In dieser Aufgabe soll zunächst zu einem gegebenen binären Suchbaum ein strukturgleicher Binärbaum erstellt werden, sodass jede Knotenbeschriftung der Bearbeitungsnummer in der Inorderreihenfolge entspricht. Mithilfe dieses so erstellten Baumes kann nun das $k$-kleinste Element im Suchbaum schnell ermittelt werden.

Ein Beispiel: zu dem Baum auf Seite 109




wird zunächst der folgende Baum erstellt




Soll nun das 3-te Element im linken Suchbaum gefunden werden, so ist die Wurzel der 4-te Knoten, das gesuchte Element ist somit im linken Unterbaum zu finden. Der Wurzel im linken Unterbaum, der 12, ist die Inordernummer 2 zugeordnet, wir suchen in dssen rechtem Unterbaum weiter. Dessen Wurzel hat die Inordernummer 3, ist somit das gesuchte Element: die 13.

- Geben Sie eine C/C++-Datenstruktur für einen binären Suchbaum oder Binärbaum an. Der Datentyp für eine Variable, die auf die Wurzel eines solchen Baums zeigt, soll dabei `bbPtr` heißen.
- Schreiben Sie eine C/C++-Funktion `bbPtr inorderKopie(const bbPtr root)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, der kopiert werden soll, und als Ergebnis den Zeiger auf die Wurzel des erstellten Baumes zurückliefert. (Wenn Sie diese Aufgabe nicht komplett lösen können, schreiben Sie alternativ eine C/C++-Funktion, die in einem Binärbaum lediglich die Knotenbeschriftung so ändert, dass diese jeweils der Nummer im Inorderdurchlauf entspricht (bis zu 5 Punkte).)

- Schreiben Sie eine weitere C/C++-Funktion
  `int baumselect(const bbPtr root, const bbPtr ioKopie, const int k)`, die als Parameter den Zeiger auf die Wurzel des binären Suchbaums bekommt, in dem gesucht werden soll, einen weiteren Zeiger auf die Wurzel des Binärbaums, der durch die obige Funktion `inorderKopie` entstanden ist, sowie eine Zahl `k`. Rückgabewert der Funktion soll das nach oben beschriebener Methode gefundene Element sein (falls `k` zu klein oder zu groß ist, so sei das Ergebnis `-1`).
- Geben Sie den Aufwand Ihres Algorithmus `inorderKopie(...)` in $O$-Notation an.
- Mit welchem Aufwand im Mittel würden Sie für einen Aufruf Ihrer Funktion `baumselect(...)` rechnen?

Beachten Sie: die Parameter sind an die Funktion zu übergeben, Sie müssen sich *nicht* um die Eingabe eines Baumes kümmern.

**Zusatzaufgabe:** Mit welchem Aufwand würden Sie rechnen, wenn im linken Suchbaum ein neuer Knoten eingefügt wird und der der rechte Baum angepasst werden muss (beachten Sie, dieser muss nicht komplett neu erstellt werden).