

Algorithmen & Datenstrukturen

Patrick Bucher

Inhaltsverzeichnis

1	Einführung	2
1.1	Was ist ein Algorithmus?	2
1.2	Was ist eine Datenstruktur?	3
1.3	Was ist ein Programm?	3
1.4	Fragen betreffend Algorithmen/Datenstrukturen	3
1.5	Beispiel: Berechnung des ggT	4
1.6	Gleichwertigkeit	5
1.7	Komplexität	5
1.7.1	Beispiel	5
1.7.2	Big-O-Notation	6
1.7.3	Abschliessende Bemerkungen	6
2	Rekursion	7
2.1	Rekursion vs. Iteration (am Beispiel der Fakultät)	7
2.1.1	Iterativer Ansatz	7
2.1.2	Rekursiver Ansatz	7
2.2	Mächtigkeit der Rekursion	8
2.2.1	Beispiel: Fibonacci-Zahl	8
2.3	Typen von Rekursion	9
2.4	Beispiel: Permutation	9
3	Collections	9
3.1	Eigenschaften von Datenstrukturen	10
3.2	Das Java Collection Framework	10
3.2.1	Grundlegende Funktionen für das Funktionieren des Collection-Frameworks	10
3.2.2	Arten der Gleichheit	11
3.3	Beispiel: Speicherverwaltung	11
3.3.1	Anforderungen an die Speicherverwaltung	11
4	Datenstrukturen	11
4.1	Eigenschaften von Datenstrukturen	11

4.2	Operationen auf Datenstrukturen	12
4.3	Statisch oder dynamisch	12
4.4	Element-Beziehungen: explizit oder implizit	13
4.5	Zugriff: direkt oder indirekt (sequenziell)	13
4.6	Aufwand von Operationen	13
4.7	Array	13
4.7.1	Binäre Suche	13
4.7.2	Einfügen	14
4.7.3	Entfernen	14
4.7.4	Empfehlungen	14
4.8	Listen	14
4.8.1	Eigenschaften von Listen	15
4.8.2	Aufwand	15
4.8.3	Modellierung	15
4.8.4	Vergleich (Liste vs. Array)	15
4.9	Stack	15
4.9.1	Aufwand	16
4.10	Queue	16
4.10.1	Aufwand	16
5	Bäume	16
5.1	Beispiel: UNIX-Dateisystem	17
5.2	Grundelemente eines Baumes	17
5.3	Kenngößen von Bäumen	17
5.4	Füllgrad eines Baumes	18
5.5	Binäre Bäume	18
5.5.1	Traversieren von binären Bäumen	18
5.5.2	Binäre Suchbäume	19
5.5.3	Operationen	19
6	Glossar	19

1 Einführung

1.1 Was ist ein Algorithmus?

Ein Algorithmus ist ein präzise festgelegtes *Verfahren zur Lösung eines Problems* bzw. einer Problemklasse; ein Lösungsverfahren (Rezept, Anleitung).

Eigenschaften eines Algorithmus:

1. schrittweises Verfahren
2. ausführbare Schritte
3. eindeutiger nächster Schritt (determiniert)

- endet nach endlich vielen Schritten (terminiert)

Beispiele für Algorithmen:

- Berechnung des ggT
- Zeichnen von Verbindungslinien
- Sortierung von Zahlen
- Finden des kürzesten Weges zwischen zwei Punkten
- Primzahltest
- Berechnung eines Integrals
- Finden einer Lösung in einem Lösungsraum

Themenbereiche von Algorithmen:

- Algorithmentheorie: Finden guter Lösungsalgorithmen für bestimmte Problemstellungen
- Komplexitätstheorie: Ressourcenverbrauch (Rechenzeit, Speicherbedarf)
- Berechenbarkeitstheorie: Was ist mit einer Maschine lösbar/nicht lösbar?

1.2 Was ist eine Datenstruktur?

Eine Datenstruktur ist ein Konzept zur *Speicherung und Organisation von Daten*. Sie ist durch die Operationen charakterisiert, welche Zugriffe und Verwaltung realisieren.

Beispiele für Datenstrukturen:

- Array: direkter Zugriff, fixe Grösse
- Liste: sequentieller Zugriff, flexible Grösse

1.3 Was ist ein Programm?

Ein Programm kombiniert Algorithmen und Datenstrukturen.

Der Ressourcenverbrauch eines Algorithmus (Laufzeit und Speicherbedarf) hängt von der Verwendung geeigneter Datenstrukturen ab.

Algorithmen operieren auf Datenstrukturen und Datenstrukturen bedingen spezifische Algorithmen.

1.4 Fragen betreffend Algorithmen/Datenstrukturen

1. Für kleine oder grosse Probleme adäquat?
2. Selber entwickeln oder aus einer Bibliothek?
3. Einfach oder schwierig zu verstehen, implementieren und warten?
4. Geringe Laufzeit mit grossem Speicherbedarf oder umgekehrt?

1.5 Beispiel: Berechnung des ggT

Gegeben sind zwei Zahlen, A und B.

1. A sei die grössere der beiden Zahlen (andernfalls tauschen).
2. Setze $A = A - B$
3. Wenn $A \neq B$: Schritt 1, wenn $A = B$: Fertig

Iterative Lösung (mit impliziter Vertauschung):

```
public static int ggT(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Iterative Lösung (mit Modulo-Operator “abgekürzt”):

```
public static int ggT(int a, int b) {
    while ((a != b) && (b != 0)) {
        if (a > b) {
            a = a % b;
        } else {
            b = b % a;
        }
    }
    return (a + b);
}
```

Rekursive Lösung:

```
public static int ggT(int a, int b) {
    if (a > b) {
        ggT(a - b, b);
    } else {
        if (a < b) {
            return ggT(a, b - a);
        } else {
            return a;
        }
    }
}
```

1.6 Gleichwertigkeit

- Alle Implementierungen führen zum Ziel und liefern die gleichen Resultate. Sie sind *gleichwertig*.
 - Die Anzahl Schleifendurchläufe, arithmetische Operationen und Methodenaufrufe – und somit Laufzeit und Speicherbedarf – unterscheiden sich jedoch.
1. Zu jeder Problemklasse gibt es verschiedene konkrete Probleme.
 2. Zu jedem konkreten Problem gibt es verschiedene Algorithmen.
 3. Zu jedem Algorithmus gibt es verschiedene Implementierungen.

Die Gleichwertigkeit von Algorithmen kann nicht bewiesen werden (Halting-Problem).

1.7 Komplexität

- Die Komplexität (oder Aufwand, Kosten) eines Algorithmus besagt, wie der Ressourcenbedarf von den Eingabedaten abhängt.
- Der Ressourcenbedarf ist eine Funktion der Eingabedaten: $R = f(E)$
 - Rechenzeit: *Zeitkomplexität*
 - Speicherbedarf: *Speicherkomplexität*
- Abhängigkeit von Eingabedaten:
 - Grösse der *Datenmenge* (z.B. Anzahl zu sortierender Elemente)
 - Grösse der *Datenwerte* (z.B. Grösse zu prüfender Primzahlen)

Bei der Komplexität eines Algorithmus geht es nicht um die exakte Rechenzeit (einer Implementierung), sondern um das *Anwachsen des Ressourcenbedarfs* in Abhängigkeit von wachsenden Eingabedaten.

1.7.1 Beispiel

Laufzeit eines Programms abhängig vom Eingabeparameter n :

```
public static void task(int n) {
    task1();
    task1();
    task1();
    task1();
    for (int i = 0; i < n; i++) {
        task2();
        task2();
        task2();
        for (int j = 0; j < n; j++) {
            task3();
            task3();
        }
    }
}
```

```

    }
  }
}

```

Annahmen:

- Die Methoden `task1`, `task2` und `task3` haben eine konstante und vergleichbare Rechenzeit und sind *nicht* vom Eingabeparameter n abhängig.
- Die Ausführungszeiten der Schleifensteuerung sind vernachlässigbar.

Berechnung: Die Rechenzeit T von `task(n)` beträgt $T = f(n) \sim 4 + 3n + 2n^2$

Folgerung:

- Für grosse n dominiert der Anteil von n^2 .
- Die Funktion ist von der Ordnung $O(n^2)$

1.7.2 Big-O-Notation

- O , das landausche Symbol, bringt zum Ausdruck, dass eine Funktion $f(n)$ höchstens so schnell wächst wie eine andere Funktion $g(n)$.
- Wird n genügend gross gewählt, unterscheiden sich $f(n)$ und $g(n)$ nur noch durch eine Konstante.

Wichtige Ordnungsfunktionen:

Bezeichnung	Notation	Beispiele
Konstant	$O(1)$	Hashing
Logarithmisch	$O(\ln(n))$	binäres Suchen
Linear	$O(n)$	Suchen in Text
$n \cdot \log(n)$	$O(n \cdot \log(n))$	schlaues Sortieren
Polynomial	$O(n^m)$	einfaches Sortieren
Exponential	$O(m^n)$	Optimierungen
Fakultät	$O(n!)$	Permutationen, Travelling Salesman

1.7.3 Abschliessende Bemerkungen

- Die Ordnung macht keine Aussage über das Verhalten bei kleinen n .
- Konstante Faktoren können bei kleinen n relevant sein.
- Die exakte mathematische Analyse vieler Algorithmen ist schwierig oder sogar unmöglich.
- Bei der Analyse muss darum differenziert werden:
 - bester Fall (best case)
 - schlechtester Fall (worst case)
 - mittlerer Fall (average case)

2 Rekursion

Viele Algorithmen und Datenstrukturen sind von Natur aus *selbstähnlich* bzw. *selbstbezüglich*.

- Der ggT von 21 und 15 ist gleich dem ggt von 21-15 und 15.
- Ein Verzeichnis enthält Daten und andere Verzeichnisse.
- Ein Ausschnitt einer Matrix, einer Liste, eines Baumes, eines Graphen ist wiederum eine Matrix, eine Liste, ein Baum, ein Graph.

2.1 Rekursion vs. Iteration (am Beispiel der Fakultät)

2.1.1 Iterativer Ansatz

Iterative Definition: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$

Iterative Implementierung:

```
public static factorial(int n) {  
    int result = 1;  
    for (int i = 0; i <= n; i++) {  
        result *= i;  
    }  
    return result;  
}
```

2.1.2 Rekursiver Ansatz

Rekursive Definition:

- Rekursionsbasis:
 - $0! = 1$
 - $1! = 1$
- Rekursionsvorschrift: $n! = n \cdot (n-1)!$
- Rekursion: Aufzeigen eines Lösungsweges, wie ein schwieriges Problem auf ein gleichartiges aber einfacheres Problem zurückgeführt werden kann.

Beispiel: $5!$ wird auf die Rekursionsbasis zurückgeführt:

```
- `5!=5*4!`  
- `5!=5*(4*3!)`  
- `5!=5*(4*(3*2!))`  
- `5!=5*(4*(3*(2*1!)))=5*(4*(3*(2*1)))=120`
```

Rekursive Implementierung:

```

public static factorial(int n) {
    if (n == 0 || n == 1) {
        // Iterationsbasis
        return 1;
    } else {
        // Rekursionsvorschrift (Rückführung)
        return n * factorial(n - 1);
    }
}

```

2.2 Mächtigkeit der Rekursion

- Rekursion und Iteration sind *gleich mächtig*.
- Die Menge der berechenbaren Problemstellungen bei Verwendung von Rekursion und Iteration ist gleich.
- Eine iterative Implementierung lässt sich immer in eine rekursive Implementierung überführen (und umgekehrt).
- Vorteile der Rekursion:
 - Rekursive Ansätze sind oft einfach und elegant.
 - Die Korrektheit rekursiver Definitionen lässt sich oft einfacher aufzeigen.
 - Es gibt rein rekursive Programmiersprachen, z.B. LISP und Prolog
- Nachteile der Rekursion:
 - Es ergeben sich schnell sehr viele Methodenaufrufe.
 - Die Programmausführung ist dadurch tendenziell langsamer.
 - Es besteht grosser Speicherbedarf auf dem Call-Stack und die Gefahr für einen Stack Overflow.

2.2.1 Beispiel: Fibonacci-Zahl

- Fibonacci-Zahlen sind in der Mathematik rekursiv definiert:
 - $f(0) = 0$
 - $f(1) = 1$
 - $f(n) = f(n-1) + f(n-2)$, für $n \geq 2$

Rekursive Implementierung:

```

public int fib(n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

```


Probleme einer rekursiven Implementierung:

- Viele rekursive Aufrufe: Komplexitätsklasse $O(2^n)$.
- Mehrfache Berechnung der gleichen Zahlen. (Lösungsansatz: Caching)
- Besser eine iterative Lösung finden!
- *Nicht jedes Problem, das sich rekursiv präsentiert, sollte rekursiv umgesetzt werden!*

2.3 Typen von Rekursion

- linear vs. nichtlinear:
 - *lineare Rekursion*: eine Methode ruft sich intern selber auf (z.B. Fakultät)
* $m() \rightarrow m()$
 - *nichtlineare Rekursion*: ein Methodenaufruf führt zu mehreren rekursiven Aufrufen (z.B. Fibonacci)
 - * nicht geschachtelt (Fibonacci): *primitiv rekursiv*, $m() \rightarrow m(); m();$
 - * geschachtelt: *nicht primitiv rekursiv*, $m() \rightarrow m(m())$
- direkt vs. indirekt:
 - *Direkte Rekursion*: eine Methode ruft sich *direkt* selber auf.
 - *Indirekte Rekursion*: eine rekursive Methode ruft sich *indirekt* selber auf, $m() \rightarrow m(n(m()))$

2.4 Beispiel: Permutation

Ein sechs Zeichen langes Passwort bestehend aus den Buchstaben von A bis F, wobei jeder Buchstabe nur einmal vorkommt, ist gesucht. Gefragt sind die *Permutationen* (alle möglichen Kombinationen) dieser Buchstaben. Es gibt $6! = 720$ mögliche Kombinationen:

- ABCDEF
- BACDEF
- ...
- FEDCBA

Lösungsansatz: Rückführung des Problems (Permutation von n Zeichen auf Permutation von einem Zeichen)!

- A kombiniert mit den Permutationen von BCDEF
 - B kombiniert mit den Permutationen von CDEF
 - * C kombiniert mit den Permutationen von DEF

3 Collections

Datenstrukturen dienen zur effizienten Speicherung und Verarbeitung von *Mengen* von Daten.

- Verarbeitung: Funktionen, die einzelne Objekte bearbeiten, nach bestimmten Objekten suchen, sie zählen, filtern, nach bestimmten Kriterien sortieren usw.

Beispiele:

- Array
- Tree
- List
- Map
- Queue
- Set
- Stack

Es gibt keine universelle Datenstruktur, die alles perfekt kann, sondern für jede Aufgabe mehr oder weniger geeignete Datenstrukturen.

3.1 Eigenschaften von Datenstrukturen

- Grösse: dynamisch/statisch
- Zugriff: direkt oder sequenziell
- Sortierung: sortiert/unsortiert, mit/ohne Ordnung
- Suche: beschleunigt?
- Geschwindigkeit (von Operationen auf die Datenstruktur): Suchen, Einfügen, Anhängen, Entfernen, Verschieben

3.2 Das Java Collection Framework

- Schnittstellen: abstrakte Datentypen
- Implementierungen: konkrete Implementierungen der Schnittstellen
- Algorithmen: (meist) polymorph implementierte Methoden zur Behandlung von Datenstrukturen, z.B.:
 - `iterator()`: sequenzieller Zugriff auf alle Elemente
 - `sort(List<E>)`: sortiert beliebige List-Implementierung

3.2.1 Grundlegende Funktionen für das Funktionieren des Collection-Frameworks

- Sortierung: `equals()`, `Comparable<T>` (beide durch Element-Klasse implementiert), `Comparator<T>` (durch Client implementiert)
- Hashing: `hashCode()` (durch Element-Klasse implementiert)

3.2.2 Arten der Gleichheit

- Typgleichheit (wenn `instanceof true` zurückliefert, Objekte Instanzen gleicher Klassen sind); lockere Form der Gleichheit
- Identität (das gleiche Objekt `==`); strengste Form der Gleichheit (gilt, wenn `equals()` nicht überschrieben wird)
- Wertgleichheit (gilt, wenn `equals()` überschrieben wurde); am häufigsten verwendete Form der Gleichheit

3.3 Beispiel: Speicherverwaltung

- Annahme (Vereinfachung): Speicher steht in linearem Adressraum zur Verfügung
- Verwaltung mittels Allokationen: jede Allokation speichert Startadresse und eine Grösse
- Problematik bei der Verschiebung von Speicherblöcken: Aktualisierung der darauf verweisenden Zeiger
- Freigegebener Speicher wird bloss als nicht besetzt markiert, aber nicht mit 0 überschrieben

3.3.1 Anforderungen an die Speicherverwaltung

- Zuverlässigkeit (keine Leaks)
- Schnell: keine lange Suche nach passendem, freien Block bei Allokation; bei Freigabe
- Zusammenfassen nebeneinanderliegender Lücken (Umgang mit Fragmentierung)
- Welche Datenstruktur eignet sich dazu am besten? (eine Tabelle bzw. Map)
- Belegungsstrategie
 - erster passender Block
 - noch kleinster, passender Block
- Freigabestrategie
 - erkennen und verbinden benachbarter, freier Blöcke
 - idealer Zeitpunkt zum Zusammenfügen
- geeignete Datenstruktur

4 Datenstrukturen

4.1 Eigenschaften von Datenstrukturen

- Datenstruktur als reine Sammlung: ungeordnete Ablage, nicht deterministische Reihenfolge (Analogie: Steinhaufen), keine Elemente dürfen verloren gehen

- Datenstruktur mit einer bestimmten Reihenfolge: Reihenfolge bleibt (Analogie: Bücherstapel)
- Datenstruktur, die Elemente (beim Einfügen) sortieren (Analogie: vollautomatisches Hochregallager)

4.2 Operationen auf Datenstrukturen

Grundlegende Operationen:

- Einfügen von Elementen
- Suchen von Elementen
- Entfernen von Elementen
- Ersetzen von Elementen: Kombination von Suchen, Entfernen und Einfügen

Operationen in Abhängigkeit von Reihenfolge oder Sortierung:

- Nachfolger
- Vorgänger
- Sortierung
- Minimum/Maximum

4.3 Statisch oder dynamisch

- statisch: feste, bei der Initialisierung definierte Grösse (Analogie: Flasche, benötigt und bietet immer gleich viel Platz)
 - Vorteile:
 - * Grösse immer bekannt (einfache Planbarkeit)
 - * einfache Implementierung
 - * einfache Adressierbarkeit (Direktzugriff auf Elemente)
 - Nachteile:
 - * Aufwändige Operationen erforderlich, wenn die Kapazität ausgeschöpft ist
 - * Verschwendung bei geringer Auslastung
- dynamisch: kann Grösse während Lebensdauer verändern und theoretisch beliebig viele Elemente aufnehmen (Analogie: Luftballon)
 - Vorteile:
 - * Man braucht sich bei der Erstellung keine Gedanken darüber zu machen, wie viele Daten gespeichert werden müssen
 - * keine Platzverschwendung (besonders zu Beginn)
 - Nachteile:
 - * Schwieriger zu implementieren
 - * Aufwändigere Operationen (ständig neue Speicherallokationen notwendig)

4.4 Element-Beziehungen: explizit oder implizit

- explizit: Elemente sind miteinander verknüpft; Elemente kennen ihre Nachbarglieder
 - Analogie: Fahrradkette
 - Beispiel: (doppelt) verkettete Liste
- implizit: Beziehungen zwischen Elementen nicht von Elementen festgehalten, sondern von übergeordneter Struktur
 - Analogie: Bücherregal
 - Beispiel: Array, Liste, Map

4.5 Zugriff: direkt oder indirekt (sequenziell)

- direkter Zugriff
 - sofort auf ein beliebiges Element zugreifen (mittels Index oder Key)
 - Beispiel: Speicheradresse im Computer
- indirekter (sequenzieller) Zugriff
 - es müssen Vorgängerelemente durchlaufen werden, um auf ein bestimmtes Element zu kommen
 - Beispiel: Magnetband, Audio-Kassette, Java Streams

4.6 Aufwand von Operationen

- variiert je nach Operation und Anzahl Elemente
- vor allem die Ordnung interessant

Beispiel: Stack konstant beim Einordnen, n^2 beim Sortieren

4.7 Array

- in Java: Sprachelement
- statisch
- implizit
- direkter Zugriff
- Reihenfolge stabil
- Aufwand
 - durchsuchen: $O(n)$
 - sortieren: $O(\log n)$

4.7.1 Binäre Suche

- in der Mitte teilen

- anhand Trennelement unterscheiden: links oder rechts weitersuchen?
- rekursiv auf ausgewählte Hälfte wiederholen
- fertig, wenn nur noch ein Element vorhanden ist

4.7.2 Einfügen

- unsortiert
 - wenn man sich letzten freien Platz merkt: $O(1)$
 - ohne diesen Trick: $O(n)$
- sortiert
 - Position suchen: $O(\log n)$
 - restliche Elemente nach rechts schieben: $O(n)$
 - Aufwand: $O(n)$

4.7.3 Entfernen

- unsortiert (fortlaufend befüllt)
 - Position suchen: $O(n)$
 - Entfernen und Lücke (mit beliebigem Element!) schliessen: $O(1)$
 - Aufwand: $O(n)$
- sortiert (fortlaufend befüllt)
 - Position suchen: $O(\log n)$
 - Entfernen und Lücke (mittels nach links durchschieben!) schliessen: $O(n)$
 - Aufwand: $O(n)$

4.7.4 Empfehlungen

Arrays sollten eingesetzt werden, wenn:

- die Datenmenge *beschränkt*, von anfang an *bekannt* und eher *klein* ist
- die Datentypen *elementar* sind, also eine bekannte und konstante Grösse haben

Sonst sind Collections, wie z.B. eine `ArrayList`, vorzuziehen.

4.8 Listen

- einfach verkettete Listen: nur vorwärts iterierbar
 - Referenz auf das erste Element (head)
 - jedes Element (jeder Node) kennt seinen direkten Nachfolger
 - das letzte Element zeigt auf `null`
- doppelt verkettete Listen: vor- und rückwärts iterierbar
 - Referenzen auf das erste (head) und das letzte (tail) Element

- jedes Element kennt seinen direkten Nachfolger und Vorgänger
- das letzte Element zeigt auf `null`

4.8.1 Eigenschaften von Listen

- dynamisch: grösser passt sich der Anzahl zu speichernder Elemente an
- explizit: Nodes “kennen” einander
- indirekter Zugriff: von head vorwärts oder von tail rückwärts durchsuchbar
- Reihenfolge definiert und konstant

4.8.2 Aufwand

- Suchen: $O(n)$ (mit Optimierung $O(\log n)$ möglich)
- Einfügen (unsortiert): $O(1)$
 - wenn es bei einer einfach verketteten Liste bei head eingefügt wird
 - wenn es bei einer doppelt verketteten Liste bei head oder tail eingefügt wird
- Einfügen (sortiert): $O(n)$
- Entfernen: $O(n)$ (wegen Suche, reines Entfernen nur $O(1)$)

4.8.3 Modellierung

- zwei Klassen:
 - Liste
 - * enthält head (und tail)
 - * Hilfsattribute (z.B. Anzahl Elemente)
 - Elemente
 - * TODO: p.27

4.8.4 Vergleich (Liste vs. Array)

- Liste besser für grosse, variierende Datenmengen
- Liste hat geringeren Einfügeaufwand
- Listen können in Java generisch implementiert werden, Arrays nicht

4.9 Stack

- Stapelspeicher
 - `push()`

TODO: p.32

4.9.1 Aufwand

Implementierung mittels Liste oder Array:

- $\text{push}(): O(1)$
- $\text{pop}(): O(1)$

TODO: p.34-35

4.10 Queue

- Warteschlange
 - $\text{enqueue}(E)$: Element am Ende anhängen
 - $E \text{ dequeue}$
 - TODO: p.39

4.10.1 Aufwand

- Implementierung mit doppelt verketteter Liste:
 - enqueue : mit $O(1)$
 - dequeue : mit $O(1)$
- Implementierung mit einem Array: als Ring Buffer!
 - Index für erstes und letztes Element “rotiert”
 - Indizes dürfen einander nicht “überholen”
 - * $\text{Index} \% \text{Arraygrösse} = \text{Position}$
 - * vor Index 0 liegt Index $n-1$
 - enqueue : mit $O(1)$ (gilt nur für Einfügen am Ende der Warteschlange!)
 - dequeue :

TODO: p.40-42

5 Bäume

- Bäume in der Informatik stehen kopfüber
- Einsatz: zwei Szenarien
 1. Daten haben eine inhärent hierarchische Struktur
 - Dateisystem
 - Stammbaum
 - Vererbungshierarchie
 2. Nicht-hierarchische Strukturen werden zum schnellen Suchen in geordneten Baumstruktur (geordnete Datenmenge) übergeführt
 - Aufwand: $O(\log n)$ bei binärer Suche (schneller als sequenzielle Suche mit $O(n)$)

- Verschiedene Arten von Baumstrukturen
 - Gemeinsamkeiten: Wurzel/Stamm
 - Unterschiede
 - * Anzahl äste (Vergabelung)
 - * Unterschiedliche Länge (Tiefe)
 - * Breite (Grad) und Höhe des Baumes sind variabel
 - Ungerichteter Baum (reine Hierarchie)
 - Out-Tree: Navigation von der Wurzel nach unten (zu den Blättern)
 - In-Tree: Navigation von den Blättern nach oben (zur Wurzel)
 - Spezialformen:
 - * Binär-Baum: TODO
 - * AVL-Baum: höhenbalancierter Binärbaum
 - * B-Baum: balancierter Baum
 - * B*-Baum: restriktivere Form des balancierten Baums
 - * Binominal-Baum: speziell strukturierter Baum

5.1 Beispiel: UNIX-Dateisystem

- Wurzelverzeichnis /
- Verzeichnisse auf erster Stufe: `etc`, `home`, `proc`, `var`, `dev`
- Knoten repräsentieren Verzeichnisse und enthalten eine Liste von Dateien und anderen Verzeichnissen
- TODO: p.10

5.2 Grundelemente eines Baumes

- Wurzel (root): hat kein übergeordnetes Element
- (innere) Knoten (node): hat $1 \dots n$ (Kinder-)knoten
- Blatt (leaf): Knoten ohne Kinder
- verbindende Pfeile werden als Kanten (edges) bezeichnet
 - gerichtet: Pfeil zeigt (bei out-tree) vom übergeordneten zum untergeordneten Knoten

5.3 Kenngrößen von Bäumen

- *Ordnung* eines Baumes: definiert maximale Anzahl Kinder pro Knoten
- *Grad* eines Knotens: bezeichnet die Anzahl Kinder eines Knotens
 - Blätter haben den Grad 0
- *Pfad* eines Knotens: Weg von der Wurzel bis zum entsprechenden Knoten
- *Tiefe* eines Knotens: seine Entfernung zur Wurzel (Zählen der Pfadelemente)
- *Niveau* eines Baumes: Menge aller Knoten mit gleicher Tiefe
- *Höhe* eines Baumes: Tiefe des Knotens mit der grössten Entfernung zur Wurzel

- *Gewicht* eines Baumes: Anzahl enthaltener Knoten

TODO: Grafik mit Beispiel für jede Grösse

5.4 Füllgrad eines Baumes

- *ausgefüllter* Baum: jeder innere Knoten hat die maximale Anzahl Kinder
 - die Ordnung der inneren Knoten ist gleich dem Grad des Baumes
- *voller* Baum: letztes Niveau linksbündig angeordnet und ausser diesem jedes Niveau die maximale Anzahl an Kindern enthält (TODO: p.22)
- *vollständiger/kompletter* Baum:
 - maximale Anzahl Knoten auf jedem Niveau
 - minimale Anzahl Niveaus für sein Gewicht
 - symmetrisch ausgeglichen

5.5 Binäre Bäume

- ein Baum mit Ordnung 2: jeder Knoten hat maximal zwei Kinder
 - linkes und rechtes Kind
- Struktur ermöglicht binäre Suche
- Traversieren

5.5.1 Traversieren von binären Bäumen

auf drei unterschiedliche Arten (rekursiv):

1. Preorder: Hauptreihenfolge (x-l-r)
 - a. verarbeite Knoten x
 - b. traversiere linken Kindknoten: `preorder(x.leftChild)`
 - c. traversiere rechten Kindknoten: `preorder(x.rightChild)`
 - d. TODO Grafik p.10
 - e. die Wurzel wird als erstes verarbeitet
2. Postorder: Nebenreihenfolge (l-r-x)
 - a. traversiere linken Kindknoten: `preorder(x.leftChild)`
 - b. traversiere rechten Kindknoten: `preorder(x.rightChild)`
 - c. verarbeite Knoten x
 - d. TODO Grafik p.12
 - e. die Wurzel wird als letztes verarbeitet
3. Inorder: symmetrische Reihenfolge (l-x-r)
 - a. traversiere linken Kindknoten: `preorder(x.leftChild)`
 - b. verarbeite Knoten x
 - c. traversiere rechten Kindknoten: `preorder(x.rightChild)`

- d. TODO Grafik p.14
- e. die Wurzel wird genau in der Hälfte verarbeitet
- f. Ausgabe in der Reihe der Sortierung

TODO: p.8

5.5.2 Binäre Suchbäume

- identisch mit binären Bäumen (Ordnung: 2)
- jeder Knoten enthält neben dem Datenelement noch einen *Schlüsselwert* (zum Sortieren oder zum Suchen)
 - Schlüsselwert aus (Teil-)Daten des Datenelements
 - Schlüsselwert aus Berechnung über (Teil-Daten) des Datenelements (z.B. `hashCode()`)

Geordneter binärer Suchbaum:

- jeder Schlüssel im linken Teilbaum eines Knotens ist kleiner als der Schlüssel des Knotens
- jeder Schlüssel im rechten Teilbaum eines Knotens ist grösser als oder gleich dem Schlüssel des Knotens
- dies ermöglicht eine effiziente *binäre Suche*

5.5.3 Operationen

- Suchen: $O(\log n)$ (2er-Logarithmus)
 - Beispiel: 1024 Elemente - 10 Entscheidungen ($2^{10}=1024$)
- Einfügen:
 1. Suchen des geeigneten Platzes zum Einfügen
 2. Eigentliches Anfügen an den gefundenen Node
 3. Die Einfügereihenfolge gibt die Struktur des Baumes vor (im Schlimmsten Fall entsteht bei Befüllen eines Baumes eine einfach verkettete Liste)
- Entfernen:
 1. Blatt: einfach entfernen
 2. innerer Knoten mit einem Kindknoten: den inneren Knoten durch den Kindknoten ersetzen
 3. innerer Knoten mit zwei Kindknoten: holy fucking jumping shitballs (p.25)
 - TODO: p.26 kleinsten (am weitesten links gelegene) Wert im rechten Teilbaum suchen

6 Glossar

- Algorithmus: präzise festgelegtes Verfahren zur Lösung eines Problems bzw. einer Problemklasse; ein Lösungsverfahren (Rezept, Anleitung)-Operator

- Datenstruktur: ein Konzept zur *Speicherung und Organisation von Daten*. Sie ist durch die Operationen charakterisiert, welche Zugriffe und Verwaltung realisieren.
- Rekursion: Aufzeigen eines Lösungsweges, wie ein schwieriges Problem auf ein gleichartiges aber einfacheres Problem zurückgeführt werden kann.