Zusammenfassung

Algorithmen und Datenstrukturen 1

Michael Wieland

Hochschule für Technik Rapperswil

13. August 2017

Mitmachen
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

Inhaltsverzeichnis

1 Generelles

ADT: Abstract Data Type
Ein abstrakter Datentyp beschreibt eine Datenstruktur generell, sodass diese in einer beliebigen Sprache umgesetzt werden kann.

Sentinel Objekte
Sind Wächter Objekte die an den Enden von z.B einer Linked List vorkommen. Ein Sentinel ist also ein Terminator einer Sequenz (Header, Trailer)

NIL Not in List

Default Initialisierung (Java)

- Primitive Datentypen: 0
- Boolsche Datentypen: false
- Objekte (auch Strings): null

String Konkatinierung
Für das Zusammensetzen von String sollte immer ein StringBuilder verwendet werden. Insbesondere bei grosser Anzahl von Teilstrings ist der Zeitunterschied enorm.

1.1 Comparator

- Die beiden Objekte a und b müssen das Interface Comparable implementieren. Ansonsten soll eine Exception geworfen werden
- compare(a, b)
  - falls a $<$ b $\Rightarrow$ negativ Wert
  - falls a $=$ b $\Rightarrow$ 0
  - falls a $>$ b $\Rightarrow$ positiver Wert

1.2 Switch Case

Folgende Datentypen können in einer Switch Anweisung verwendet werden

- byte, short, char, int
- Enumerations
- String, Character, Byte, Short, Integer

```
switch(n) {
   case 1: n + 1; break;
   case 2: n + 2;
   // also executed for case 2, because of missing break;
   case 2: n + 3;
}
```

2 Mathematische Grundlagen

2.1 Allgemeine Summenformel

$$\sum_{i=1}^{n} a_i = \frac{n(a_n + a_1)}{2} = \frac{n(n+1)}{2}$$

2.2 Geometrische Reihe

Muss bei Exponentiellem Wachstum verwendet werden.

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

2.3 Fakultät

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n = \prod_{k=1}^{n} k$$

$$n! \cdot (n+1) = (n+1)!$$

wobei

$$0! = 1$$

2.4 Beispiel: Arithmetische Folgen

Bestimmen Sie da n-te Glied (a_n) der Folge 1,5,9,13,17:

Rekursiv: $a_1 = 1$ und $a_n = a_{n-1} + 4$

Iterativ: $a_1 + \sum_{i=2}^{n} 4$

Explizit: $a_1 + incr \cdot (n-1) = 1 + 4(n-1) = 4n - 3$

2.5 Beispiel: Summenformelbeweis

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

2.5.1 Verankerung n=1

linke Seite = rechte Seite?

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

$$1 = 1 \Rightarrow \text{erfüllt}$$

2.5.2 Induktionsschritt n = n + 1

linke Seite:

$$\sum_{n}^{2n-1} \Rightarrow \sum_{n+1}^{2n+1} \tag{2}$$

Die beiden Summenformeln ausschreiben und Unterschiede markieren.

$$\begin{aligned} &\frac{1}{n} + \frac{1}{n+1} + \frac{1}{n+2} + ... + \frac{1}{2n-2} + \frac{1}{2n-1} \\ &\frac{1}{n+1} + \frac{1}{n+2} + ... + \frac{1}{2n-1} + \frac{1}{2n} + \frac{1}{2n+1} \end{aligned} \tag{3}$$

Unterschiede an bestehendes Anhängen

$$\sum_{n}^{2n-1} \frac{1}{i} - (\frac{1}{n}) + (\frac{1}{2n}) + (\frac{1}{2n+1}) \tag{4}$$

rechte Seite:

$$\sum_{1}^{2n-1} \Rightarrow \sum_{1}^{2n+1} \tag{5}$$

Die beiden Summenformeln ausschreiben und Unterschiede markieren.

$$\begin{aligned} &\frac{(-1)^2}{1} + \frac{(-1)^3}{2} + \frac{(-1)^4}{3} + ... + \frac{(-1)^{2n-1}}{2n-2} + \frac{(-1)^{2n}}{2n-1} \\ &\frac{(-1)^2}{1} + \frac{(-1)^3}{2} + \frac{(-1)^4}{3} + ... + \frac{(-1)^{2n}}{2n-1} + \frac{(-1)^{2n+1}}{2n} + \frac{(-1)^{2n+2}}{2n+1} \end{aligned} \tag{6}$$

Unterschiede an bestehendes Anhängen

$$\sum_{1}^{2n-1} \frac{(-1)^{i+1}}{i} + (\frac{(-1)^{2n+1}}{2n}) + (\frac{(-1)^{2n+2}}{2n+1}) \tag{7}$$

Zusammenführen
Nun können die Summenformeln weggelassen werden, da diese gemäss Induktionsbehauptung

äquivalent sind.

$$\begin{aligned}
\frac{1}{2n} + \frac{1}{2n+1} - \frac{1}{n} &= \frac{(-1)^{2n+1}}{2n} + \frac{(-1)^{2n+2}}{2n+1} \\
\frac{1}{2n} + \frac{1}{2n+1} - \frac{1}{n} &= \frac{\overbrace{(-1)^{2n+1}}^{-1}}{2n} + \frac{\overbrace{(-1)^{2n+2}}^{1}}{2n+1} \\
\frac{1}{2n} - \frac{1}{n} &= \frac{-1}{2n} \\
\frac{1}{2n} + \frac{1}{2n} - \frac{1}{n} &= 0 \\
\frac{2}{2n} &= \frac{1}{n} \\
\frac{1}{n} &= \frac{1}{n}
\end{aligned} \tag{8}$$

2.6 Beispiel: Rekursive Abschätzung

Gegeben sei folgende rekursive Abschätzung des Zeitverhaltens eines Algorithmus:

$$T(n) = 16 \cdot T(\frac{n}{4}) - 33 \cdot n + 15$$
$$T(1) = S \text{ (Startwert)} = 10$$

Fragestellung: Ist $T(n) \in \mathcal{O}(n)$ oder $T(n) \in \mathcal{O}(n^2)$?

Lösungs-Ansatz $T(n) = a \cdot n^2 + b \cdot n + c$

Auflösen von Variablen

$$\begin{aligned}
T(\frac{n}{4}) \Rightarrow T(n) &= a \cdot n^2 + b \cdot n + c \\
T(\frac{n}{4}) &= \frac{a \cdot n^2}{16} + \frac{b \cdot n}{4} + c
\end{aligned} \tag{9}$$

Einsetzen und Vereinfachen Nun kann $T(\frac{n}{4})$ in die rekursive Laufzeitabschätzung eingesetzt, und mit der beispielhaften Laufzeit von $\mathcal{O}(n^2)$ gleichsetzt werden.

$$\begin{aligned}
16 \cdot (\overbrace{\frac{a \cdot n^2}{16} + \frac{b \cdot n}{4} + c}^{T(\frac{n}{4})}) - 33 \cdot n + 15 &= \overbrace{a \cdot n^2 + bn + c}^{\mathcal{O}(n^2)} \\
\Leftrightarrow a \cdot n^2 + 4bn + 16c - 33n + 15 &= a \cdot n^2 + bn + c
\end{aligned} \tag{10}$$

Behandlung der quadratischen Teile n^2

$$\begin{aligned}
a \cdot n^2 &= a \cdot n^2 \\
a &= a
\end{aligned} \tag{11}$$

Behandlung der linearen Teile n

$$\begin{aligned} 4bn - 33n &= bn \\ 4b - 33 &= b \\ -33 &= -3b \\ b &= 11 \end{aligned} \tag{12}$$

Behandlung der konstanten Teile c

$$\begin{aligned} 16c + 15 &= c \\ 15 &= -15c \\ c &= -1 \end{aligned} \tag{13}$$

Laufzeit angeben Es sei der Startwert $T(1) = 10$ gegeben:

$$\begin{aligned} &10 = n^2 \cdot a + n \cdot b + c \\ &10 = 1 \cdot a + 1 \cdot 11 - 1 \\ &a = 0 \Rightarrow n^2 \text{ Teile sind Null und daher ist die Laufzeit} = \underline{\underline{\mathcal{O}(n)}} \end{aligned} \tag{14}$$

3 Beweisen

Der Beweis durch vollständige Induktion sei folgend anhand einem Beispiel erklärt

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1} \text{ wobei } n \geq 1$$

3.1 Induktionsverankerung (n = 1)

Ist die linke Seite gleich der rechten Seite?

1. Linke Seite: $\frac{1}{1(1+1)} = \frac{1}{2}$
2. Rechte Seite: $\frac{1}{2}$

Die Induktionsverankerung ist somit erfüllt.

3.2 Induktionsschritt (n = n + 1)

3.2.1 Induktionsannahme

Unsere Annahme ist, dass die Induktionsverankerung für n gilt. Man schreibt noch einmal die Verankerung hin.

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

3.2.2 Induktionsbehauptung

Somit muss sie die Annahme auch für $n = n + 1$ gelten. Man ersetze alle n mit $n + 1$

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

3.2.3 Induktionsbeweis

$$\underbrace{\frac{n}{n+1}}_{\text{rechte Seite der Annahme}} + \underbrace{\frac{1}{(n+1)(n+2)}}_{\text{k = n+1 aus Annahme}} = \underbrace{\frac{n+1}{n+2}}_{\text{rechte Seite der Behauptung}}$$

$\Leftrightarrow n(n+2) + 1 = (n+1)^2$

$\Leftrightarrow n^2 + 2n + 1 = n^2 + 2n + 1$

$\rightarrow$ Erfüllt, quod erat demonstrandum

4 Landau-Symbole

4.1 Big Oh Notation $\mathcal{O}(x)$

- Bei der Big Oh Notation konzentriert man sich immer auf den Worst Case.
- Die Big Oh Notation lässt sich die Laufzeit und Speicherverbrauch eines Algorithmus beschreiben
- Dabei wird immer ein konkreter Fall angeschaut. (z.B die Laufzeit einer Schleife unter Bedingung i > 1)
- f(n) ist $\mathcal{O}$(g(n)) falls f(n) asymptotisch kleiner oder gleich wie g(n) ist

4.1.1 Vorgehen

1. primitive Operationen Zähle, rsp. wie oft eine Anweisung ausgeführt wird.
2. Konstante Faktoren können weggelassen werden
3. Gibt es mehrere Potenzen (n^2, n^5) gilt nur die höchste. Alle anderen können ebenfalls weggelassen werden.
4. $\mathcal{O}$(x) aufschreiben. z.B $\mathcal{O}(n^2)$ oder $\mathcal{O}(n)$

4.2 Big Omega Notation $\Omega(x)$

- f(n) ist Ω(g(n)) falls f(n) asymptotisch grösser oder gleich wie g(n) ist

4.3 Big Theta Notation $\Theta(x)$

- f(n) ist Θ(g(n)) falls f(n) asymptotisch gleich wie g(n) ist

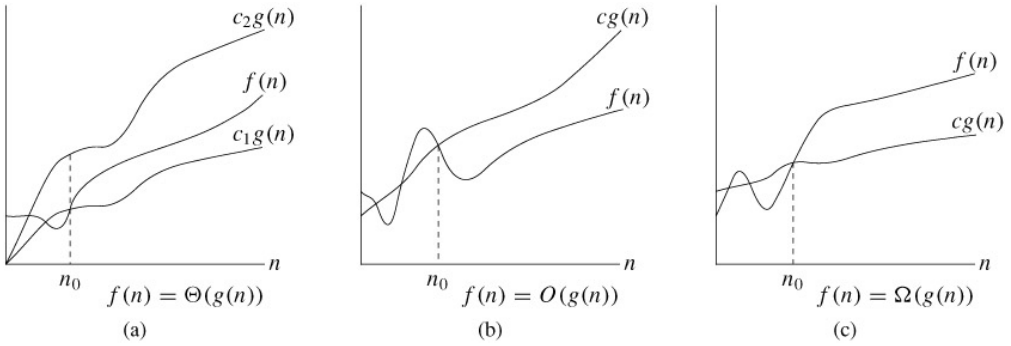

Abbildung 1: Big Theta / Big Oh / Big Omega

5 Analyse von Algorithmen

Algorithmus

Ein Algorithmus ist ein Schritt für Schritt Vorgehen zum Lösen eines Problems mit endlichem Zeitaufwand.

5.1 Laufzeiten

1. $\mathcal{O}(1)$ = konstant
2. $\mathcal{O}(\log(n))$ = logarithmisch
3. $\mathcal{O}(n)$ = linear
4. $\mathcal{O}(n \log(n))$ = n-Log-n
5. $\mathcal{O}(n^2)$ = Quadratisch
6. $\mathcal{O}(n^3)$ = Qubisch
7. $\mathcal{O}(2^n)$ = Exponentiell
8. $\mathcal{O}(n!)$ = Fakultät

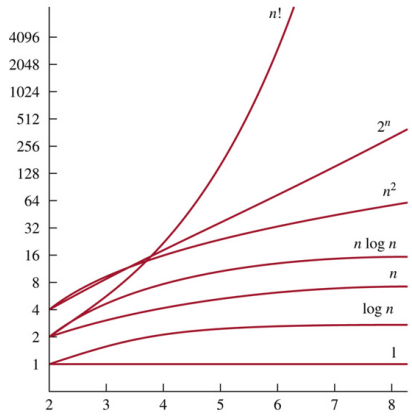

Abbildung 2: Laufzeiten

6 UML: Unified Modeling Language

Aggregation und Komposition

Beide sind "Ist-Teil-von-Ganzem" Beziehung, wobei die Komposition stärker bindet. Wird bei der Komposition der Parent gelöscht, müssen auch alle Childs gelöst werden.

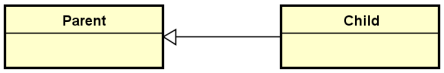

Abbildung 3: Generalisierung

Abbildung 4: Implementierung / Dependency

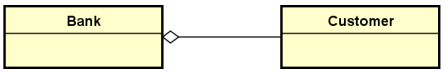

Abbildung 5: Aggregation

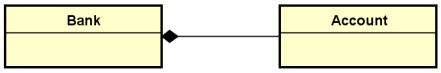

Abbildung 6: Komposition

7 Design Pattern

7.1 Adapter Pattern

Das Entwurfsmuster findet in erster Linie Anwendung, wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle nicht der benötigten Schnittstelle entspricht. Adapter ermöglichen die Zusammenarbeit von Klassen, die ohne nicht zusammenarbeiten könnten, weil sie inkompatible Schnittstellen haben. Es gibt zwei Varianten von Adaptern:

7.1.1 Objektadapter

- Auch als Wrapper bekannt
- Verwendet Komposition, welche zur Laufzeit hinzugefügt wird. Die Adapter Klasse besitzt also eine Instanzvariable vom Typ des Adaptee
- Kapselt besser, da die Methoden des Adaptee nicht sichtbar sind
- Alle Methoden die der Adapter zur Verfügung soll, müssen im Target Interface implementiert sein.
- Wird verwendet wenn man dem Client nur Methoden mit abgeänderter Adaptee Funktionalität zur Verfügung stellen möchte. Die restliche Adaptee Funktionalität soll aber versteckt bleiben.
- Der Objektadapter wird wie folgt verwendet:

```
1       public class Client {
2          // Adapter implements Target
3          private Target myAdapter = new Adapter();
4
5          public Client() {
6             myAdapter.request();
7          }
8       }
```

7.1.2 Klassenadapter

- Verwendet Mehrfachvererbung welche bereits zur Compilezeit fixiert ist. Unter Java muss ein Interface und einer abstrakte Klasse verwendet werden, da Java keine Mehrfachvererbung unterstützt.
- Trägt die Methoden des Adaptess nach aussen
- Wird verwendet wenn eine zusätzliche Methode implementiert werden soll und man aber immer noch auf die Methoden des Adaptee zugreifen soll.
- Der Klassenadapter wird wie folgt verwendet:

```
1    public class Client {
2       // Adapter extends Adaptee implements Target
3       private Target myAdapter = new Adapter();
4
5       public Client() {
6          myAdapter.request();
```

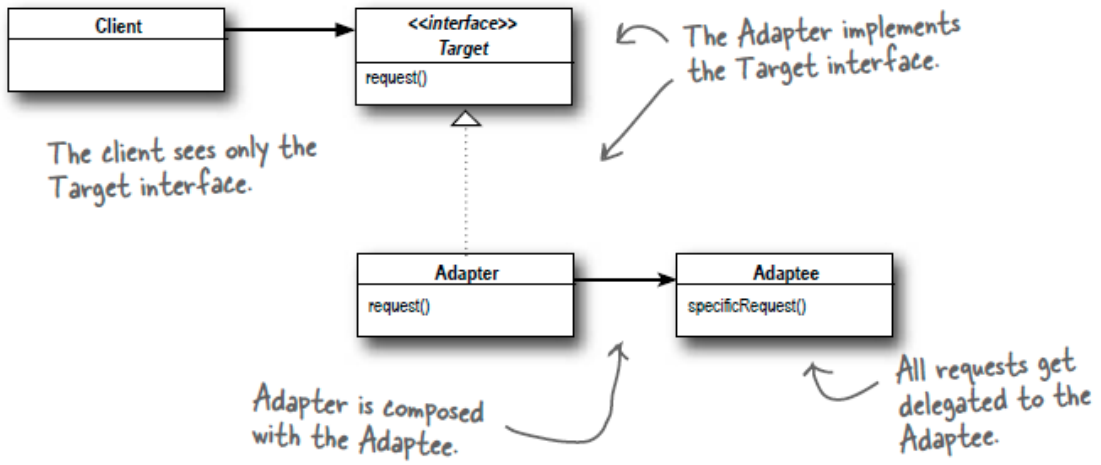

Abbildung 7: Objekt Adapter Pattern

```
7             myAdapter.adapteeMethod();
8         }
9     }
```

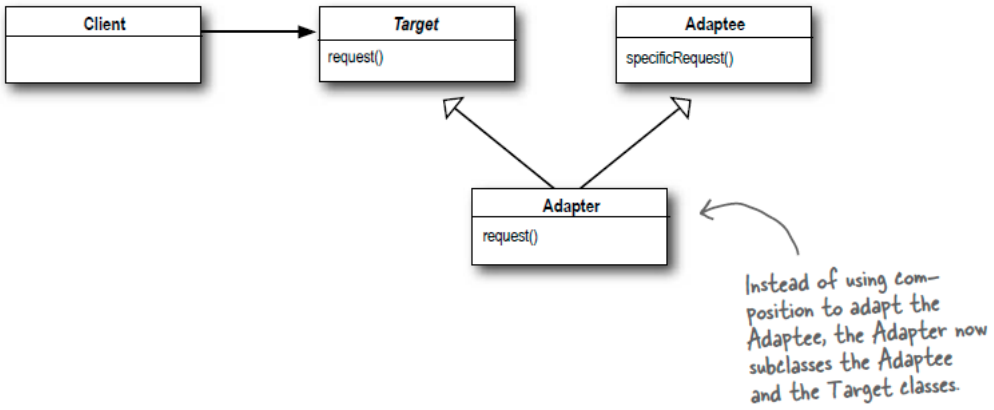

Abbildung 8: Klassen Adapter Pattern

7.2 Template Method Pattern

- Die gemeinsamen Teile werden in einer abstrakten Klasse implementiert und in den Subklassen bei Bedarf verfeinert.
- Die abstrakte Klasse besteht aus unveränderlichen Methoden welche zusätzlich abstrakte Methoden (Hooks) aufruft, welche in der Subklasse überschrieben werden müssen.
- Das Pattern verwendet das Hollywood Prinzip: ”Don’t call us, we call you”

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}

public class Football extends Game {
   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

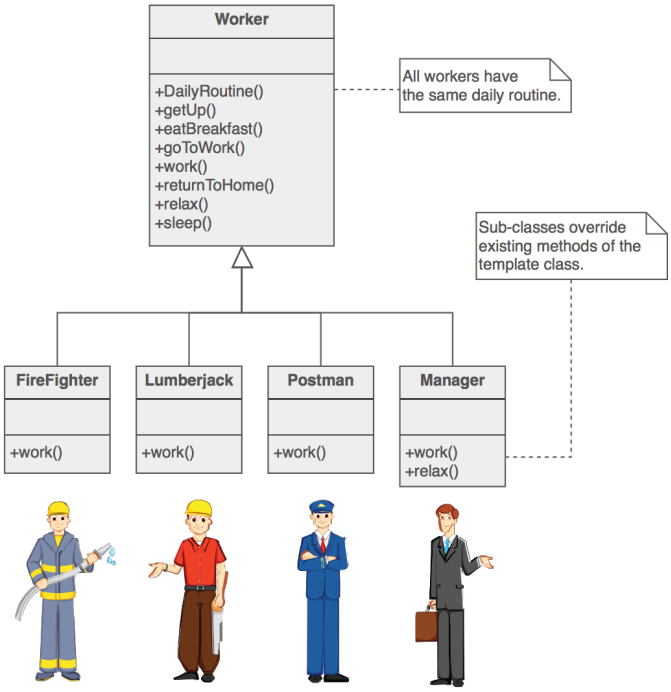

Abbildung 9: Template Method Pattern

8 Rekursion

8.1 Rekursion vs Iteration

- Iteration brauch kaum Speicher. Die Iterationsvariablen müssen einmal alloziert und können anschliessend nur noch inkrementiert werden.
- Rekursion muss immer wieder einen neuen Stack anlegen. Es droht die Gefahr eines Stack Overflows!
- Iteration ist schneller
- Alle Rekursionen können als Iteration ausgedrückt werden.
- Es gibt aber Probleme die nur sehr schwer iterativ gelöst werden können.

8.2 Lineare Rekursion

Eine lineare rekursive Methode ruft sich solange auf bis sie den BaseCase / Abbruchkriterium erreicht. Die Rekursion legt die Zwischenresultate auf den Stack und rechnet diese nach dem Ankommen beim Base Case rückwärts zusammen.

8.2.1 Fibonacci Folge

```
1 // Return fibonacci sequence (0 1 1 2 3 5 8 13 ...)
2 public static int fibonacci(int number){
3    // Base case
4    if(number <= 1){
5       return number;
6    }
7    return fibonacci(number-1) + fibonacci(number -2);
8 }
```

8.2.2 Divide & Conquer

1. Ein grosses Problem in mehrere Teile zerlegen bis ein kleines Problem trivial zu lösen ist (z.B Binär sortieren)
2. Base Case ist wenn nur noch ein Element vorig ist
3. Dieser Lösungsansatz wird für viele bekannte Algorithmen verwendet. So zum Beispiel Fibonacci, Quicksort, Euklid und Tower of Hanoi.

8.2.3 Tail Rekursion / Endrekursion

Bei einer Tail Rekursion muss die letzte Operation ein rekursiver Aufruf sein. Bei einer ”normalen” Rekursion steigt der Speicherplatzverbrauch linear mit der Rekursionstiefe. Da bei Tail Rekursionen aber das Resultat beim erreichen des Base Cases vorhanden ist, muss nur Speicherplatz für die Übergabe der Methodenparameter reserviert werden. Der restliche Speicherplatz kann wieder freigegeben werden. Somit ist der Speicherplatzverbrauch bei einer Tail Rekursion unabhängig von der Rekursionstiefe. Compiler wandlen Endrekursionen im Rahmen eines Optimierungsschrittes sogar direkt in die iterative Form um.

8.2.4 Rekursives Summieren

```
int sum(int i, int sum) {
   if(i == 0) {
      return sum;
   }
   return sum(i-1, i + sum);
}
```

8.3 Binäre Rekursion

Bei der binären Rekursion existieren zwei rekursive Aufrufe in allen nicht-terminalen (kein Base Case) Aufrufen.

8.3.1 Tower of Hanoi

$$Anzahl_{moves} = 2^n - 1$$

wobei

n Anzahl Disks

```
// Tower of Hanoi
public class TowerOfHanoi {
   public static void main(String[] args) {
      int amountOfDisks = 3;
      doTowers(amountOfDisks, 'A', 'B', 'C');
   }

   public void solve(int n, String start, String auxiliary, String end) {
       if (n == 1) {
          System.out.println("Disk-" + n + " from " + start + " to " + end);
       } else {
          solve(n - 1, start, end, auxiliary);
          System.out.println("Disk-" + n + " from " + start + " to " + end);
          solve(n - 1, auxiliary, start, end);
       }
    }

}
```

9 Arrays

- Random Access: Man kann auf jedes beliebige Element mit $\mathcal{O}(1)$ zugreifen
- Keine dynamische Grösse
- In Java speichern Arrays von Objekten nur die Referenz auf das Objekt und nicht das Objekt selbst (Ausnahme: primitive Datentypen)
- Nützliche Array Methoden:

```
1  // Vergleichen
2  Arrays.equals(object[] a1, object[] a2);
3
4  // Sortieren
5  Arrays.sort(object[] a1);
6
7  // Umkopieren
8  System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

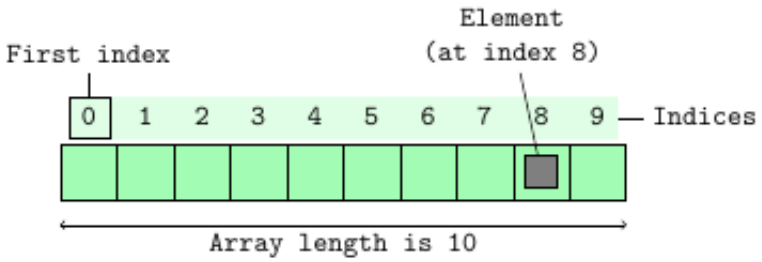

Abbildung 10: Array

10 Linked List

- Die Linked List ist eine einfach verkettete Liste. Man hat also immer nur Zugriff auf das nächste Element
- Jeder Knoten besitzt ein Element (das effektive Objekt) und einen Link zum nächsten Knoten
- Dynamische Grösse
- Kein umkopieren beim Hinzufügen und Entfernen
- Allozierter Speicher entspricht genau den Nutzdaten
- Zugriff auf das letzte rsp. mittlere (bei Doubly Linked List) sehr langsam $\Rightarrow \mathcal{O}(n)$
- Jeder Knoten besteht aus einem Pärchen bestehend aus dem Element und dem Link zum nächsten Knoten.
- Start und Ende der Linked List ist speziell gekennzeichnet. Die leere Linked List hat also mindestens zwei Elemente (Header und Trailer Node).
- Die Java Implementierung der Linked List ist eine Doubly Circular Linked List (wegen dem Deque Interface)

```
1 // Durch alle Elemente loopen
2 for (Element e = header.next; e != trailer; e = e.next)
```

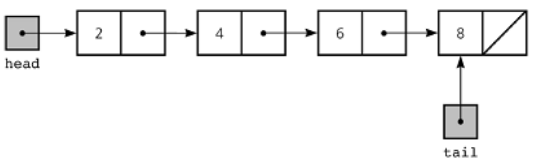

Abbildung 11: Linked List

10.1 Singly Linked List

- Beim Einfügen am Anfang einer Singly Linked List verweist das neue Element auf den aktuellen Head. Zusätzlich wird das Head Flag beim neuen Element gesetzt und beim alten Head entfernt.
- Beim Einfügen am Ende einer Singly Linked List verweist das neue Element auf null und das alte Tail verweist auf das neue Element. Zusätzlich wird das Tail Flag beim neuen Element gesetzt und beim alten Tail entfernt.
- Singly Linked Lists haben den Nachteil, dass man immer von vorne nach hinten durch iterieren muss. $\mathcal{O}(n)$

10.2 Doubly Linked List

- Jeder Knoten besteht aus einem Triple, bestehend aus dem Element (Wert) und einem Link nach vorne einen nach hinten. (prev, next)
- Die leere Liste hat mindestens zwei Knoten (Head, Tail). Head und Tail müssen im Konstruktor verbunden werden. (next, prev)

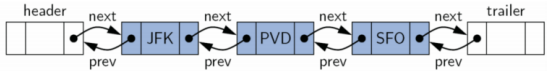

Abbildung 12: Doubly Linked List / Doppelt verkettete Liste

10.3 Circularly Linked List

Bei einer Circulary Linked List ist das Tail mit dem Head verbunden

11 Lists

```java
public interface List<E> {
   size();

   boolean isEmpty();

   // Gibt das Element an der Stelle i zurueck ohne es zu entfernen
   E get(int i);

   // Fuegt ein Element an der Stelle i in die Liste ein und verschiebt alle
       nachfolgenden Elemente um eine Stelle
   void add(int i, E element)

   // Ersetzt das Element beim Index i und liefert das alte Element zurueck
   void set(int i, E element);

   // Entfernt das Element an der Stelle i und gibt es zurueck
   E remove(int i);
}
```

11.1 ArrayList

- Die Klasse ArrayList speichert die Daten intern in einem primitiven Array
- Die Arraylist hat eine variable Grösse und vergrössert das interne Array (fixe Grösse) selbständig
- Der Zugriff mit get(), set(), size(), isEmpty() ist $\mathcal{O}(1)$
- Beim Einfügen und Entfernen muss immer Platz geschaffen/entfernt werden. (add(i, e), remove(i)) Im schlimmsten Fall müssen alle Elemente verschoben werden $\Rightarrow \mathcal{O}(n)$. Der absolute Worst Case ist i=0, dann dann alle n Elemente verschoben werden müssen.

11.1.1 Strategien dynamische Arraygrösse

Es gibt zwei Ansätze wie das interne Array vergrössert wird:

1. Inkrementelle Strategie: Erweitern der Grösse um eine Konstante k $\Rightarrow \mathcal{O}(n^2)$
2. Verdoppelung Strategie: Verdoppeln der Arraygrösse $\Rightarrow \mathcal{O}(n)$

11.2 Positional-List / Node-List

- Jeder Knoten besitzt einen Vorgänger und einen Nachfahren. Dies erlaubt es Beziehungen zwischen den Knoten zu ziehen. (Ausnahme: Head und Tail)
- Jeder Knoten ist ein Position Objekt, welches eine Referenz auf das Element hält.
- Benötigt für eine List mit n Elementen $\mathcal{O}(n)$ Speicher
- Alle Operationen des Node-List ADT benötigt $\mathcal{O}(1)$ Zeit, sofern man sich an der Position p befindet.
- Die Operation getElement() benötigt $\mathcal{O}(1)$

- Suche nach einer Position benötigt $\mathcal{O}(n)$
- Eine Doubly Linked List stellt eine einfache Implementierung des Positional List ADT dar.

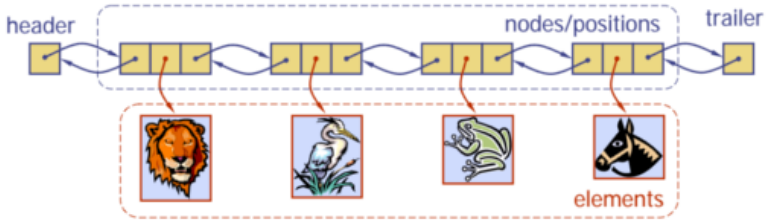

Abbildung 13: Positional List

```
1  public interface Position<T> {
2     Position<T> first();
3     Position<T> last();
4
5     Position<T> addFirst(T element);
6     Position<T> addLast(T element);
7
8     // Gibt die Position vor oder nach der gegeben Position zurueck
9     Position<T> before(Position<T> position);
10    Position<T> after(Position<T> position);
11
12    // Setzt die Position vor oder nach der gegeben Position
13    Position<T> addBefore(Position<T> p, T e) throws IllegalArgumentException;
14    Position<T> addAfter(Position<T> p, T e)throws IllegalArgumentException;
15
16    E set(Position<T> p, T e) throws IllegalArgumentException;
17    E remove(Position<T> p) throws IllegalArgumentException;
18
19    Iterator<T> iterator();<
20    Iterable<Position<T>> positions();
21 }
```

11.3 Skip List

- Eine Skip Liste besteht aus einer Menge von Sub Listen
- jede Subliste ist sortiert
- die Enden jeder Subliste ist mit eindeutigen Sentinels markiert. Gibt es mehrere 'leere' Sublisten mit zwei Sentinels werden alles bis auf einen gelöscht.
- Ist besonders geeignet für Multithreading
- In einer perfekten Skip Liste ist die Höhe = $log_2(n)$
- Jeder 2^k Knoten hat einen Zeiger auf den 2^k entfernten Knoten auf Niveau k
- Beim Einfügen wird mit einer Wahrscheinlichkeit von 50% entschieden ob der Turm erhöht wird oder nicht.

- Der Suchpfad wird in einer temporären Liste abgespeichert
- Suchen, Einfügen und Löschen läuft mit $\mathcal{O}(\log(n))$

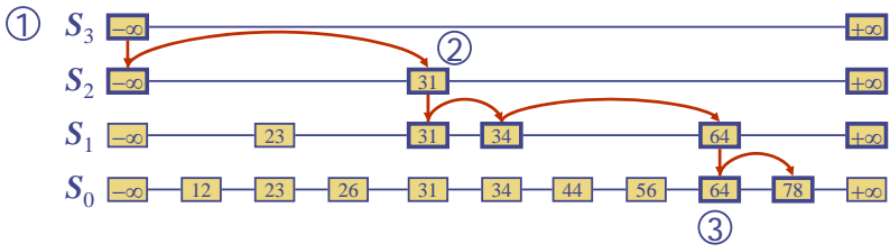

Abbildung 14: Skip List

11.3.1 Vorgehen Einfügen

Die Elemente einer Skipliste werden mit der Hilfe einer Zufallssequenz eingefügt. (50% Chance für Inkrementierung der Ebene)

1. Erstes Element aus der Liste entnehmen und in Level S_0 eintragen. Die Elemente werden auf der x-Achse nach ihren Werten geordnet (links=$-\infty$ und rechts=$+\infty$)
2. Aus der Zufallssequenz nächste Zahl entnehmen und durchstreichen
   a) 1 (Kopf) $\Rightarrow$ Turm erhöhen mit aktuellem Element
   b) 0 (Zahl) $\Rightarrow$ nächstes Element aus der Liste nehmen und auf Level S_0 einfügen.

11.3.2 Vorgehen Suche

1. Wenn das gesuchte Objekt grösser oder gleich dem nächsten Objekt ist, wird scanForward() ausgeführt. Die Operation "scanForward" iteriert auf der aktuellen Liste nach vorne. Falls scanForward() $+\infty$ zurückgibt wird dropDown() ausgeführt.
2. Wenn das gesuchte Objekt grösser oder gleich dem aktuellen Objekt ist, wird dropDown() ausgeführt. Die Operation "dropDown" geht eine Ebene tiefer.
3. Falls man auf der untersten Ebenen angelangt ist und ein weiteres Mal dropDown() ausgeführt wird, muss null zurückgegeben werden.

12 Sets

- Enthält nie Duplikate
- Ist unsortiert

```
1 public interface Set<E> {
2    add(e)
3    remove(e)
4    contains(e)
5    iterator();
6    addAll();
7    retainAll();
8    removeAll();
9 }
```

12.1 Multiset

- Erlaubt Duplikate

13 Maps

- Eine Map ist eine durchsuchbare Struktur von Key-Value-Entries.
- Pro Key ist nur ein Entry erlaubt!

```
1  public interface Map<K,V> {
2     // Gibt Value oder null fuer den Key k zurueck
3     get(k);
4     // Ersetzt den Wert fuer den Key k. Falls kein Entry fuer k existiert wird ein
          neues Angelegt und null zurueckgegeben. Ansonsten wird das ersetzte letzte
          Wert retourniert
5     put(k, v);
6     // Gibt den Wert fuer k zurueck und loescht das Entry in der Map. Gibt es keinen
          Eintrag fuer k wird null zurueckgegeben.
7     remove(k);
8     // Liefert iterierbare Collection aller Schluessel
9     keySet();
10    // Liefert iterierbare Collection aller Werte
11    values();
12    // Liefert iterierbare Collection aller Eintraege
13    entrySet();
14 }
```

13.1 Map als List

Eine Map kann ganz einfach mit einer doppelt verketten Liste implementiert werden. Dieser Ansatz ist jedoch nur bei kleinen Maps sinnvoll, da die Funktionen put(), get() und remove() mit $\mathcal{O}(n)$ laufen. $\mathcal{O}(n)$ daher, da die ganze Liste auf Duplikate durchsucht werden muss. Dabei wird die Key-Value Entries als Elemente der Liste abgespeichert.

Sentinel Trick Der gesuchte Knoten wird in einem ersten Schritt als Trailer der List hinzugefügt. Danach kann man über die Liste iterieren ohne immer prüfen zu müssen, ob man das Ende der Liste erreicht hat.

```
1  private MapEntry<K, V> find(Object key) {
2     MapEntry<K, V> sentinel = new MapEntry<K, V>((K)key, null);
3     list.addLast(sentinel);
4     Iterator<MapEntry<K, V>> it = list.iterator();
5     while (true) {
6        MapEntry<K, V> e = it.next();
7        K thisKey = e.getKey();
8        if (thisKey.equals(key) || thisKey != null && thisKey.equals(key)) {
9           list.removeLast(); // remove sentinel
10          if (e.equals(sentinel)) {
11             return null;
12          } else {
13             return e;
14          }
15       }
16    }
17 }
```

13.2 Hashtabelle

Eine Hashtabelle besteht immer aus den folgenden zwei Elementen

1. Einer Hashfunktion wessen resultierender Hashwert als Index für das Array verwendet wird.
2. Einem Array mit einer fixen Grösse

13.3 HashMap

Bei einer HashMap werden die Keys mit einer Hashfunktion gehashed. Die resultierenden Hashwerte dienen als Index in dem dahinter liegenden Array. Eine gute Hashfunktion verteilt die Einträge so gut, dass keine Kollisionen auftreten. (gute Streuung) Wenn der generierte Hashcode grösser wie der grösste Index des Arrays ist, wird der Hashcode modulo Array Länge gerechnet. (Kompressionsfunktion) Gibt es eine Kollision (zwei Einträge auf einem Index) wird eine verkettete Liste an der Position des Indexes angelegt.

13.4 Kompressionsfunktion

- Als Grösse für die Hashtabelle wird aufgrund der Zahlentheorie oft eine Primzahl verwendet.
- Zur Kompression wird meist der resultierende Hashwert modulo Hashtabellengrösse verwendet.

13.5 Kollisionsbehandlung

geschlossene Adressierung
Bei einer Kollision werden weitere Objekte in einer verketteten Liste abgelegt. Diese Variante erfordert eine weitere Datenstruktur, was zusätzlicher Speicherverbrauch bedeutet.

offene Adressierung
Bei einer Kollision wird einfach der nächste offene Platz gesucht. Dabei gibt es diverse Varianten wie die "nächste" offene Stelle gesucht wird. (linear nach vorne, linear nach hinten, (linear Probing) quatrischer Ansatz, etc.)

Doppeltes Hashing
Hierbei wird bei einer Kollision einen zweiten Hashwert berechnet. Das Resultat des zweiten Hashwertes entspricht dann den Anzahl Stellen um welche verschoben wird. $Hashwert = (h_1(k) + c \cdot h_2(k)) mod N$ (c = Anzahl Kollisionen)

13.5.1 Löschen bei linearer Sondierung

Soll ein Datensatz gelöscht werden, so kann dies die Sondierungsfolge für einen andere Datensatz unterbrechen. Deshalb darf man die Einträge nicht wirklich löschen sondern nur als gelöscht markieren. ⇒ Spezielles Dummy Objekt ablegen. (DEFUNCT Objekt)

13.6 Polynom Akkumulation / Horner Schema

Man zerlege die Bits eines Schlüsseles in mehrere Teile fixer Länge und berechne davon das Polynom $p(z) = a_0 + a_1 z + a_2 z + ... + a_{n-1} z^{n-1}$. Das Polynom p(z) kann in $\mathcal{O}(n)$ mithilfe des Horner Schemas berechnet werden. Dieses Verfahren wird von Java verwendet.

13.6.1 String Hashcode

Der Hashcode für String wird unter Java wie folgt berechnet

$$hashcode = stringArr[0] \cdot 31^{n-1} + stringArr[1] \cdot 31^{n-2} + ... + stringArr[n-1]$$

13.7 Performance

Hashing ist grundsätzlich sehr schnell ist im Idealfall $\mathcal{O}(1)$ wenn keine Kollisionen auftreten. Ansonsten kann das Laufzeitverhalten zu $\mathcal{O}(n)$ führen.

13.8 Multimap / Dictionary

- Pro Key gibt es mehrere Werte
- Auch bekannt als Dictionary
- Es gibt zwei Ansätze wie eine Multimap implementiert werden kann
  1. Ein Schlüssel verweisst auf eine Collection mit Werten zu einem Key
  2. Anpassen der darunterliegenden Datenstruktur, das mehrere Key gespeichert werden können

14 Queues

- Eine einfache Queue ist FIFO.
- Das Einfügen erfolgt am Ende der Queue, das Entfernen am Anfang
- z.B Round Robin Scheduler

```
1  public interface Queue<E> {
2
3  public int size();
4
5  public boolean isEmpty();
6
7  // Fuegt ein Element am Ende der Queue ein
8  public void enqueue(E element);
9
10 // Entfernt und gibt das Element am Anfang der Queue zurueck.
11 public E dequeue();
12
13 // Liefert das erste Element ohne es zu entfernen
14 public E first();
15 }
```

14.1 Ringer Buffer / Array basierte Queue

Queues sind häufig als Ringpuffer mit je einem Zeiger auf Anfang (In-Pointer) und Ende (Out-Pointer) implementiert. Die Besonderheit des Ringpuffers ist, dass er eine feste Größe besitzt. Dabei zeigt der In-Pointer auf das erste freie Element im Array, das den Ringpuffer repräsentiert, und der Out-Pointer auf das erste belegte Element in dem Array. Im Unterschied zum Array werden die ältesten Inhalte überschrieben, wenn der Puffer voll ist und weitere Elemente in den Ringpuffer abgelegt werden.

$$Index_{rear} = (Index_{front} + Number_{elements} \bmod (Array_{length}))$$

14.2 Listen Basierte Queue / Node Queue

```
1  Node<E> head;
2  Node<E> tail;
3  int numberOfElements;
```

14.3 Double Ended Queue / Deque

- Bidirektionale Queue
- Das Einfügen und Entfernen erfolgt am Anfang oder am Ende
- Wird am einfachsten intern mit einer Doubly Linked List implementiert

```
1  addFirst(Element e);
2  addLast(Element e);
3  Element removeFirst();
4  Element removeLast();
```

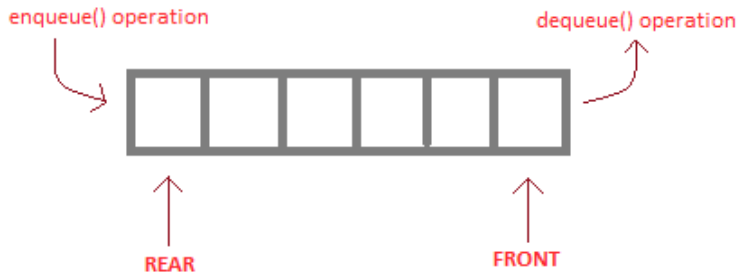

Abbildung 15: Queue

14.4 Priority Queues

- Eine sortierte Queue
- Eine Priority Queue ist eine Collection aus Entries, bestehend aus Key Value Paaren.
- Die interne Datenstruktur ist nach dem Key der Entries geordnet
- Mittels removeMin() können die Elemente sortiert entfernt werden.
- Mithilfe der Priority Queue kann ein Set aus vergleichbaren Elementen sortiert werden.

  Selection Sort Schnelles Einfügen ($\mathcal{O}(1)$), langsames herausnehmen, da die Werte beim Herausnehmen sortiert werden. ($\mathcal{O}(n)$)

  Insertion Sort Langsames einfügen ($\mathcal{O}(n)$), schnelles herausnehmen ($\mathcal{O}(1)$), da die Werte beim Einfügen sortiert werden.

  In-Place Implementierung Bei einem In-Place Algorithmus wird keine zusätzliche Datenstruktur verwendet, als jede die für die Speicherung der Daten benötigt wird.

```
1 //fuegt ein Entry mit Schluessel k und Wert v ein
2 insert(k,v);
3 // entfernt und liefert die Entry mit dem kleinsten Schluessel
4 removeMin();
5 // gibt die Entry mit dem kleinsten Schluessel zurueck ohne diese zu entfernen
6 min();
7 size();
8 isEmpty();
```

14.5 Adaptierbare Priority Queue

- Bei einer normalen Priority Queue kann ein beliebiger Node nicht einfach geändert werden. Man müsst dazu alle Nodes bis zum gesuchten Node entfernen
- Die adaptierbare PQ implementiert deshalb drei weitere Methoden:

```
1     // Entfernt das Element und gibt es zurueck
2     remove(e);
3     // Der Schluessel des Element e wird durch k ersetzt
4     replaceKey(e, k);
5     // Der Wert des Elements e wird durch v ersetzt
6     replaceValue(e, v);
```

- Eine Location Aware Entry weiss über seine Position innerhalb der Datenstruktur bescheid. Dazu hat jedes Knotenelement eine Rückreferenz auf den Knoten
- Die Listen basierte Implementierung verwendet dabei eine Positional List
- Bei der Heap Implementierung zeigt jeder Knoten auf das Werte Objekt und dieses hat wiederum eine Back Reference auf den Knoten. Der Back Pointer müssen bei up-, und downheaps() aktualisiert werden und sind deshalb optional.

15 Stack

- LIFO Prinzip: Das zuletzt eingefügte Element wird zuerst herausgenommen.
- Man hat immer nur auf das oberste Element Zugriff
- Die JVM ist eine Stack Maschiene. Sie speichert aktive Methoden in der Reihenfolge ihres Aufrufs auf dem Stack.

```
1  public interface Stack<E> {
2
3     public boolean empty();
4
5     // Top des Stacks ohne entfernen des obersten Elements
6     public E top();
7     public E peek(); // Java Util
8
9     // Entfernt das oberste Element
10    public E pop();
11
12    // Legt ein Element zuoberst auf den Stack
13    public void push(E elem);
14 }
```

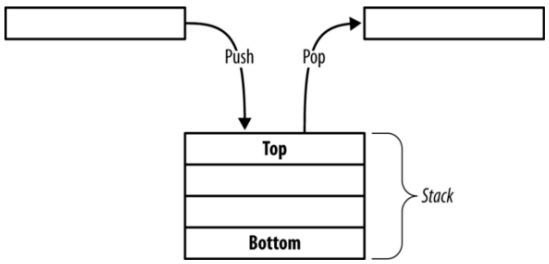

Abbildung 16: Stack

15.1 Array basierter Stack

- Jede Operation läuft mit $\mathcal{O}(1)$
- Es wird eine zusätzliche Variable für den Top of Stack benötigt
- Die maximale Grösse des Arrays ist fix. Der Stack muss bei bedarf erweitert und verkleinert werden.

15.2 Listen basierter Stack

- top() ⇒ first();
- push() ⇒ addFirst(e);
- pop() ⇒ removeFirst();

16 Tree

- Hierarchische Struktur wobei die Wurzel zuoberst ist
- Es gibt eine Eltern Kind Relation zwischen zwei Knoten

k-Baum Ein Baum mit k Kindknoten pro Node

Wurzel / Root Elternknoten

Interner Knoten Knoten mit min. einem Child

```
1     public computeInternNodes(Node<T> root) {
2        if (root == null || root.left == null && root.right == null) {
3           return 0;
4        } else {
5           return 1 + computeInternNodes(root.left) + frage(root.right);
6        }
7     }
```

Externer Knoten / Blattknoten Knoten ohne Childs:

$$Anz_{\text{externe Knoten}} = Anz_{\text{interne Knoten}} \cdot (Anz_{\text{Kind Knoten pro Baum}} - 1) + 1$$

Vorgängerknoten Parent

Tiefe Anzahl Vorgänger (nach oben)

- Wenn Wurzel Knoten: Tiefe = 0
- Ansonsten: Hole rekursiv die Tiefe des Parents und zähle 1 dazu

Höhe Anzahl Ebenen der Nachfolger (nach unten). Die Höhe gibt die Anzahl Ebenen des Baumes an.

- Externe Knoten haben die Höhe 0
- Hole rekursiv die maximale Höhe aller Children und zähle 1 dazu

Subtree Baum aus einem Knoten und seinen Nachfolger

Vorgängerknoten Parent

Siblings Zwillingsknoten

```
1  public interface Tree {
2     // Zugriffsmethoden
3     Position<E> root()
4     Position<E> parent(Position<E> p)
5     PositionList<E> children(Position<E> p)
6
7     //Abfragemethoden
8     int numChildren(Position<E> p) // nur die direkten Kinder! (Eine Ebene tiefer)
9     boolean isInternal(Position<E> p) // mit child
10    boolean isExternal(Position<E> p) // ohne child
11    boolean isRoot(Position<E> p)
12
13    //Hilfsmethoden
14    int size();
15    boolean isEmpty();
16    int iterator();
17    PositionList<E> positions();
18 }
```

16.1 Speicherverfahren

16.1.1 Linked List

Jeder Knoten wird von einer Klasse representiert die aus den folgenden drei Elementen besteht:

- Eine Referenz auf den Knoten Wert
- Eine Referenz auf den Parent Knoten
- Eine Referenz auf auf eine List von Child Knoten. Wenn es sich um einen Binary Tree handelt, wird auf die Liste verzichtet. Stattdessen wird die Referenz auf den linken und rechten Knoten direkt gespeichert.

16.1.2 Array

Das folgende Konzept funktioniert nur für den Binärbaum

1. Der Root Knoten kommt auf den Index 1
2. Der Linke Kind Knoten kommt auf den Index: $2 \cdot Index_{parent}$
3. Der Rechte Kind Knoten kommt auf den Index: $2 \cdot Index_{parent} + 1$

16.2 Traversierung

Gestartet wird immer beim Root, aufgeschrieben wird aber nur gemäss der Euler Tour Traversierung. Dabei zeichnet man startend links vom Parent Node eine Umrandung um den ganzen Tree und zieht bei jedem Knoten einen Strich in eine bestimmte Richtung:

Preorder / Strich nach Links Ein Node wird vor seinen Nachfolgern besucht, wobei zuerst der linke Node und danach der rechte Node abgearbeitet wird. ($ParentNode \Rightarrow LeftNode \Rightarrow RightNode$)

Postorder / Strich nach Rechts Ein Node wird nach seinen Nachfolgern besucht ($LeftNode \Rightarrow RightNode \Rightarrow ParentNode$). Diese Variante kann für das berechnen von arithmetischen Audrücken verwendet werden.

```
1   calcRecursive(n) {
2      if (isExternal(n)) {
3         return n.element();
4      } else {
5         x = calcRecursive(leftChild(n));
6         y = calcRecursive(rightChild(n));
7         o = parseOperator(v);
8         return x o y;
9      }
10  }
```

Inorder / Strich nach Unten Ein Knoten wird **nach** seinem linken Subtree und **vor** seinem rechten Subtree besucht. ($LeftNode \Rightarrow ParentNode \Rightarrow RightNode$) Diese Variante kann für das ausgeben von arithmetischen Ausdrücken verwendet werden.

```
printExpression(n) {
if (hasLeft(n)) {
   print('(');
   printExpression(left(n));
}
print(n.element());
if (hasRight(n)) {
   printExpression(right(n));
   print(')');
}
```

Breath First / Level Traversierung Es werden zuerst alle Nodes einer Ebenen besucht bevor man zu einer tieferen Ebenen voranschreitet. Die Nodes werden dabei von links nach rechts abgearbeitet.

```
Queue<TreeNode> queue = new LinkedList<BinaryTree.TreeNode>() ;
   public void breadth(TreeNode root) {
      if (root == null)
         return;
      queue.clear();
      queue.add(root);
      while(!queue.isEmpty()){
         TreeNode node = queue.remove();
         System.out.print(node.element + " ");
         if(node.left != null) queue.add(node.left);
         if(node.right != null) queue.add(node.right);
   }
}
```

16.3 Binärbaum

Vollständiger Binärbaum
Bei einem vollständigen Binärbaum sind alle Level bis auf das letzte vollständig aufgefüllt. Zusätzlich muss das letzte Level von links nach rechts aufgefüllt sein. Es gibt maximal ein Knoten mit nur einem Kind, welcher ein linker Knoten sein muss.

Echter Binärbaum (full/proper)
Jeder interne Knoten besitzt genau zwei Kindknoten

Balancierter Baum
Bei einem Balancierten Baum haben alle Subtrees auf jeder Ebene die selbe Höhe

- Die Position eines einmal eingefügten Knotens wird nicht mehr verändert
- Jeder interne Knoten besitzt höchstens zwei Kinder. Bei echten Binärbäumen müssen es immer genau zwei Kindknoten sein
- Die Kinder eines Knoten sind ein geordnetes Paar (links, rechts ⇒ sortiert). Die Ordnung ist dabei Definitionssache
- Jeder Subtree ist wiederum ein Binärbaum
- Anwendungsfälle sind:
  - Arithmetische Ausdrücke: Jeder Subtree stellt ein Klammerausdruck dar, wobei der Parent die Operation und die beiden Kinder die Zahlen/Variablen representieren.

- Entscheidungsprozesse: Jeweils links oder auch rechts ist immer die selbe Entscheidung.
- Binäre Suche

```java
public interface BinaryTree<E> extends Tree<E> {
   // Ein Binaerbaum besitzt zusaetzliche Methoden
   Position<E> left(Position<E> p);
   Position<E> right(Position<E> p);
   Position<E> sibling(Position<E> p);
}
```

16.4 Postorder Calculator

Die Umgekehrt Polnische Notation verwendet Postorder Calculation nach folgendem Prinzip:

Algorithm 1: evalExpr(v)

1: **if** $isExternal(v)$ **then**
2: **return** $v.element()$
3: **else**
4: $x \leftarrow evalExpr(leftChild(v))$
5: $y \leftarrow evalExpr(rightChild(v))$
6: $o \leftarrow$ Operator bei v
7: **return** $xparse(o)y$
8: **end if**

17 Heaps

Ein Heap ist ein vollständiger Binärbaum (von links her aufgefüllt), der in seinen Knoten Schlüssel speichert, wobei jeder Kindknoten einen grösseren oder gleichen Key als sein Parent haben muss. Das kleinste Element ist somit immer der Root-Knoten. Ist gibt auch sogennante MAX-Heaps wobei der Root Knoten das grösste Element ist.

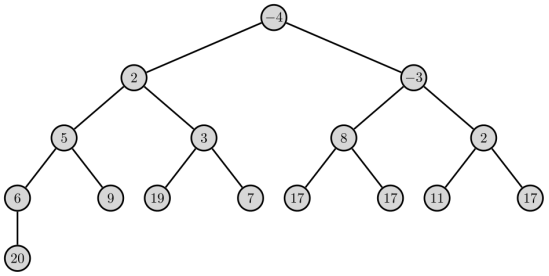

Abbildung 17: MAX-Heap in Baum und Array Ansicht

17.1 Heaps als Priority Queue

- Jeder Knoten ist ein paar aus Key-Value
- Der letzte Knoten wird speziell markiert
- Neue Knoten werden immer nach dem letzten Knoten eingefügt
- Solange der neue Knoten kleiner wie sein Parent ist, switched die Funktion upheap() der Parent und den Child
- Die Funktion upheap läuft mit $\mathcal{O}$(log n), da sie von der Höhe abhängt.
- Die Höhe ist ebenfalls $\mathcal{O}$(log n)
- Mit der Methode removeMin() kann das kleinste Element entfernt werden. Dies ist immer der Root Knoten
  - Zuerst wird der Root Knoten entfernt und mit dem letzten Knoten ersetzt
  - Danach wird der letzte Knoten gelöscht
  - Schliesslich stellt die Methode downHeap die Ordnung wieder her, indem die Knoten von oben nach unten gemäss ihrer Grösse vertauscht werden.
- Die Methode downHeap() läuft mit $\mathcal{O}$(log n)
- Der Heap muss von oben nach unten und von links nach rechts immer sortiert sein. Der Grösste Key ist immer ganz unten links!

17.2 Heap Sort / Bottom Up Konstruktion

- Läuft mit $\mathcal{O}$(n log n)
- Beim Heap Sort wird die Bottom Up Kontruktion verwendet. Der Heap wird also von unten nach oben aufgebaut. Dabei werden immer zwei Teilheaps (mit $2^i - 1$ Elementen $\Rightarrow$ i = Phase) zu einem grösseren Heap (mit $2^{i+1} - 1$ Elementen) zusammengeführt. $\Rightarrow$ benötigt $\mathcal{O}(n)$ um den Heap Bottom Up aufzubauen

18 Itertor

Ein Iterator besteht grundlegend aus den beiden Methoden next() und hasNext()

```
1 // Prueft ob es ein naechstes Element gibt (Cursor). Zeigt immer zwischen zwei
      Elemente
2 hasNext();
3 // Gibt das naechste Element in der Liste zurueck
4 next();
5
6 // Nur fuer Doubly Linked List Iteratoren
7 hasPrevious()
8 previous()
```

Es gibt zwei grundlegende Varianten

18.1 Snapshot Iterator

Es wird eine Kopie der Ausgangsdatenstruktur genommen. Änderungen auf beeinflussen die Ausgangsdatenstruktur nicht. Da zuerst die Datenstruktur kopiert werden muss, läuft diese Variante mit $\mathcal{O}(n)$.

18.2 Lazy Iterator

Dies ist die meist verbreitete Variante wobei die Iterationen auf Original Datenstruktur durchgeführt werden. Strukturänderungen an der Datenstruktur durch einen anderen Prozess können die Iteration verunmöglichen. (ConcurredModificationException) Da nichts umkopiert werden muss, sind die Kosten dafür gering.

18.3 Iterator vs For-Schleife

- Die Variante mit dem Iterator läuft mit $\mathcal{O}(n)$ da der Iterator die zugrunde liegende Datenstruktur kennt
- Die Variante mit einer For Schleife und der get(i) Funktion läuft mit $\mathcal{O}(n^2)$ da bei der Get Funktion noch einmal die ganze Liste bis zum i-ten Element durchlaufen werden muss.

19 Algorithmen

19.1 Binäre Suche

Die binäre Suche verwendet den Divide and Conquer Algorithmus

```
public boolean binarySearch(int[] data, int taget, int low, int high) {
   if (low > high)
      return null;
   } else {
      int mid = (low + high) / 2;
      if (target == data[mid]) {
         // Hit
         return true;
      } else if (target < data[mid]) {
         return binarySearch(data, target, low, mid -1);
      } else {
         return binarySearch(data, target, mid + 1, high);
      }
   }
}
```

Doubly Linked List basierter Ansatz

```
public Integer binarySearch(Integer key) {
   Node node = head;
   // amount of compareTo() used for direction
   int compare = 1;
   // Empty list = 0
   int len = (noOfNodes + 1);
   while (len > 0) {
      len = len / 2;
      if (len == 0) {
         i = -1;
         for (; i < len; i++) {
            if (compare > 0) {
               node = node.next();
            } else {
               node = node.prev();
            }
         }
         if ((node == head) || (node == tail)) {
            return null;
         }
         compare = key.compareTo(node.key);
         if (compare == 0) {
            return node.value;
         }
      }
      return null;
   }
}
```

19.2 Rekursives Array umdrehen

Dieser Algorithmus ist ein Beispiel für eine Tail Recursion. Das Resultat liegt nach dem Erreichen des Base Cases vor.

```java
public class ReverseArray {
  public int[] reverseArray(int[] a, int i, int j) {
     int temp=a[j];
     a[j] = a[i];
     a[i] = temp;
     reverseArray(a, i+ 1, j - 1);
  }
}
```

19.3 Rekursives Ausgeben einer Linked List

```java
public static void printReverse(Node<String> head) {
   if (head.next == null) {
      System.out.println(head.element);
   } else {
      // Beachte ob zuerst ausgegeben werden soll und dann rekursiver Aufruf oder
          umgekehrt
      printReverse(head.next);
      System.out.println(head.element);
   }
}
```

19.4 Rekursives Divide & Conquer

Nachdem das Maximum der linken Hälfte bestimmt wurde, wird mit der rechten Hälfte entsprechend weiterverfahren.

```java
public char maximum(char[] w, int start, int end) {
   if(start==end) {
      return w[start];
   }

   int middle = (start + end) / 2;
   char max1 = maximum(w, s, middle);
   char max2 = maximum(w, middle + 1, end);

   if(max1 < max2) {
      return max2;
   }
   return max1;
}
```

19.5 Rekursives Quadrieren

Die Rekursive Methode läuft mit $\mathcal{O}(\log(n))$

```java
public double power(double base, int exponent) {
   if (exponent == 0) {
      return 1;
   }

   double temp;
   if (exponent % 2 != 0) {
      temp = power(base, (exponent - 1) / 2);
      return base * temp * temp;
   } else {
```

```
11          temp = power(base, exponent / 2);
12          return temp * temp;
13       }
14    }
```

19.6 Klammer Matching

```
1  public static boolean parenthesisCheck(String text) {
2     Stack<Character> stack = new Stack<>();
3     for (int i = 0; i < text.length(); i++) {
4        if(text.charAt(i) == '(') {
5           stack.push(text.charAt(i));
6        } else if (text.charAt(i) == ')') {
7
8           // Empty stack or invalid parenthesis
9           if (stack.isEmpty() || stack.pop().equals(text.charAt(i))) {
10             return false;
11          }
12       }
13    }
14
15    if (stack.isEmpty()) {
16       // all parenthesis matched
17       return true;
18    } else {
19       return false;
20    }
21 }
```

19.7 Insertion Sort

Array basierter Ansatz

Beim Insertion Sort wird mit zwei Schleifen gearbeitet. Die äussere started beim Array Index 1 (zweites Objekt) und iteriert über alle Elemente des Arrays. Die innere Schleife nimmt initial den Iteration-Index der äusseren Schleife und vergleicht rückwärts die beiden Elemente auf Unterschiede.

```
1  public static void InsertionSort( int [] data) {
2     for (int i = 1; i < num.length; i++) { // Start with 1 (not 0)
3        int current = data[i];
4        int j = i;
5        while(j > 0 && data[j - 1] > current) {
6           data[j] = data[j - 1];
7           j--;
8        }
9        data[j] = current;
10    }
11 }
```

19.8 Loops in Link List erkennen

Loops können mit dem ”Floyd’s cycle finding algorithm”, welcher auch unter dem Namen ”Tortoise and hare algorithm” bekannt ist, erkannt werden. Dabei lässt man ein Element doppelt so schnell wie ein anderes durch die Liste gehen. Wenn das schnellere, dass langsamere einholt, gibt es eine Loop.

```
boolean hasLoop(Node first) {
   if(first == null) {
      return false;
   }

   // create two references
   Node slow, fast;

   // make both refer to the start of the list
   slow = fast = first; .

   while(true) {
      // single hop
      slow = slow.next;

      if(fast.next != null) {
         // two hops
         fast = fast.next.next;
      } else {
         // next node null => no loop.
         return false;
      }
      // if either hits null..no loop
      if(slow == null || fast == null) {
         return false;
      }
      // if the two ever meet...we must have a loop
      if(slow == fast) {
         return true;
      }
   }
}
```

20 Datenstrukturen und Algorithmen im Vergleich

20.1 Array vs. List

- Array haben schnelleren Zugriff auf den Index i
- Listen sind schneller beim Entfernen von Objekten
- Arrays sind schneller beim Einfügen an einem bestimmten Index
- Listen sind schneller beim Einfügen nach/vor einem bestimmten Objekte
- Listen erweitern sich dynamisch.
- Arrays müssen bei add() und remove() erweitert, verkleinert werden

Operation	Array	List
size(), isEmpty()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
atIndex(i), get(i)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
first(), last(), prev(), next()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
set(p, e)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
set(i, e)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
add(i, e), remove(i)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
addFirst(e), addLast(e)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
addAfter(p, e), addBefore(p, e)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
remove(p)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
indexOf(p)	$\mathcal{O}(n)$	$\mathcal{O}(n)$

20.2 Insertion vs Selection Sort

	Datenstruktur	Best Case	Worst Case
Selection Sort	Array	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection Sort	Doubly Linked List	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	Array	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	Doubly Linked List	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$