

---

# AuD - Zusammenfassung

Moritz Gerhardt

---

## 1 Inhaltsverzeichnis

---

<b>1</b>	<b>Inhaltsverzeichnis</b>	<b>1</b>
<b>2</b>	<b>Was ist ein Algorithmus?</b>	<b>2</b>
<b>3</b>	<b>Laufzeitanalyse</b>	<b>3</b>
3.1	$O$ Notation . . . . .	3
3.2	$\Omega$ Notation . . . . .	5
3.3	$\Theta$ Notation . . . . .	7
<b>4</b>	<b>Sortieren</b>	<b>8</b>
4.1	Sortierproblem . . . . .	8
4.2	Insertion Sort . . . . .	9
4.3	Bubble Sort . . . . .	11
4.4	Merge Sort . . . . .	13
4.5	Quicksort . . . . .	15
4.6	Radix Sort . . . . .	17
<b>5</b>	<b>Grundlegende Datenstrukturen</b>	<b>19</b>
5.1	Stacks . . . . .	19
5.2	Queues . . . . .	20
5.3	Linked List . . . . .	22
5.4	Binary Search Tree . . . . .	24
<b>6</b>	<b>Fortgeschrittene Datenstrukturen</b>	<b>27</b>
6.1	Red-Black Tree . . . . .	27
6.2	AVL Trees . . . . .	35
6.3	Splay Trees . . . . .	40
6.4	Binary Heap Trees . . . . .	44
6.5	B-Tree . . . . .	48
<b>7</b>	<b>Probabilistische Datenstrukturen</b>	<b>55</b>
<b>8</b>	<b>Graphen Algorithmen</b>	<b>56</b>

---

## 2 Was ist ein Algorithmus?

---

Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe. Dabei sollte ein Algorithmus im allgemeinen folgende Voraussetzungen erfüllen:

1. Bestimmt:

- Determiniert: Bei gleicher Eingabe liefert der Algorithmus gleiche Ausgabe.  
⇒ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
- Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.  
⇒ Gleiche Schritte, Gleiche Zwischenstände.

2. Berechenbar:

- Finit: Der Algorithmus ist als endlich definiert. (Theoretisch)
- Terminierbar: Der Algorithmus stoppt in endlicher Zeit. (Praktisch)
- Effektiv: Der Algorithmus ist auf Maschine ausführbar.

3. Anwendbar:

- Allgemein: Der Algorithmus ist für alle Eingaben einer Klasse anwendbar, nicht nur für speziellen Fall.
- Korrekt: Wenn der Algorithmus ohne Fehler terminiert, ist die Ausgabe korrekt.

---

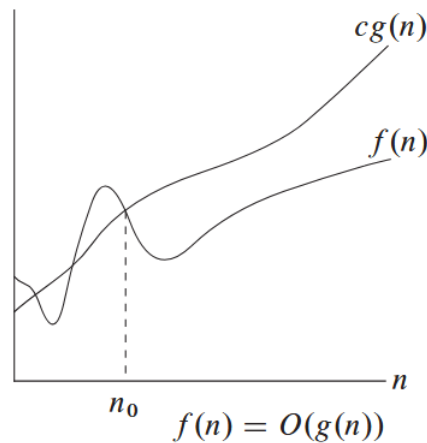
## 3 Laufzeitanalyse

---

### 3.1 O Notation

---

Die O-Notation wird grundsätzlich für *Worst-Case* Laufzeiten verwendet. Sie gibt also eine obere Schranke an, die der Algorithmus im schlechtesten Fall erreicht. Dabei wird oft zwischen Big O-Notation und Little o-Notation unterschieden. Ein Graph zur Repräsentation der O-Notation ist hier zu sehen:



#### 3.1 (a) Big-O Notation

Mathematische Definition:

$$O(g(n)) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

Es existieren die positiven Konstanten  $c$  und  $n_0$ , sodass für alle  $n \geq n_0$  gilt, dass  $f(n) \geq 0$  und  $f(n) \leq c \cdot g(n)$ . Das bedeutet, dass die Funktion  $f(n)$  für  $n \rightarrow \infty$  den gleichen Wachstumsfaktor hat wie die Funktion  $g(n)$ . Einfache Berechnung findet wie folgt statt (anhand vom Beispiel  $f(n) = 5n^2 + 2n$ ):

1. Finde den Term mit dem höchsten Wachstumsfaktor ( $5n^2$ )
2. Konstanten werden weggelassen ( $n^2$ )
3. Demnach ist  $f(n) = O(n^2)$

Dies kann man dann im Rückschluss so anwenden: Um die Konstanten  $c$  und  $n_0$  zu finden, wird die obige Gleichung benutzt:

1. Simplifiziere die Ungleichung  $5n^2 + 2n \leq c \cdot n^2$  zu  $5 + \frac{2}{n} \leq c$
2. Da  $n \geq n_0$  kann man die Gleichung für  $n \geq 1$  auflösen um die Konstanten  $c$  und  $n_0$  zu finden.  
 $\implies 5 + \frac{2}{1} = 7 \leq c \implies c \geq 7$
3. Dementsprechend kann man dann die Konstanten  $c = 7$  und  $n_0 = 1$  auswählen.

### 3.1 (b) Little-o Notation

Mathematische Notation:

$$O(g(n)) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\}$$

Es existieren die positive Konstanten  $c$  und  $n_0$ , sodass für alle  $n \geq n_0$  gilt, dass  $f(n) \geq 0$  und  $f(n) < c \cdot g(n)$ . Little-o Notation unterscheidet sich also von Big-O Notation nur oberen Schranke. Während bei Big-O der Wachstumsfaktor beider Funktion gleich sein kann ( $f(n) = c \cdot g(n)$ ), gilt bei Little-o, dass der Wachstumsfaktor der Funktion  $f(n)$  kleiner ist als der Wachstumsfaktor der Funktion  $g(n)$ .

Einfache Berechnung findet analog zu Big-O wie folgt statt (anhand vom Beispiel  $f(n) = 5n^2 + 2n$ ):

1. Finde den Term mit dem höchsten Wachstumsfaktor ( $5n^2$ )
2. Konstanten werden weggelassen ( $n^2$ )
3. Demnach ist  $f(n) = o(n^2)$

Hier muss allerdings noch geprüft werden, ob der Wachstumsfaktor der Funktion  $f(n)$  kleiner ist als der Wachstumsfaktor der Funktion  $g(n)$ . Wenn ja, ist die Little-o Notation korrekt für  $g(n)$ .

Um zu zeigen, dass  $f(n) = o(g(n))$ :

1. Finde den Limes des simplifizierten Ausdrucks  $\frac{f(n)}{g(n)}$ , der die Wachstumsrate der Funktion  $f(n)$  zur Wachstumsrate der Funktion  $g(n)$  vergleicht.
  - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{5n^2 + 2n}{n^2} = \lim_{n \rightarrow \infty} 5 + \frac{2}{n} = 5$   
 $\Leftarrow \frac{2}{n}$  für  $n \rightarrow \infty = 0$
2. Da der Limes  $\neq 0$  ist, bedeutet das, dass der Wachstum von  $f(n)$  nicht geringer ist als der von  $g(n)$ . Deshalb müssen wir ein Polynomgrad hochgehen, weswegen  $f(n) = o(n^3)$  sein muss.
3. Um nun die Konstanten  $c$  und  $n_0$  zu finden müssen wir einfach  $\frac{f(n)}{g(n)}$  auflösen
  - $\frac{5n^2 + 2n}{n^3} < c$
  - $\frac{5}{n} + \frac{2}{n^2} < c$
  - $\frac{5}{n} < c$ , da  $\frac{2}{n^2}$  für  $n \rightarrow \infty$  schneller abfällt als  $\frac{5}{n}$
  - Für  $c = 1$  muss dann  $n_0 > 5$  sein und kann somit als  $n_0 = 6$  gewählt werden.

### 3.1 (c) Rechenregeln

Sind sowohl für *Big - O* als auch *Little - o* gültig

- **Konstanten:**

$$f(n) = a \text{ mit } a \in \mathbb{R}_{>0} \implies f(n) = O(1)$$

Ist die Funktion konstant, so ist die Komplexität  $O(1)$ .

- **Skalare Multiplikation:**

$$f(n) = O(g(n)) \implies a \cdot f(n) = O(g(n))$$

Multiplikation der Funktion ändert die Komplexität nicht.

- **Addition:**

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \implies f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$$

Die Komplexität der Summe zweier Funktionen ist der Maximalwert der Komplexität der beiden Funktionen.

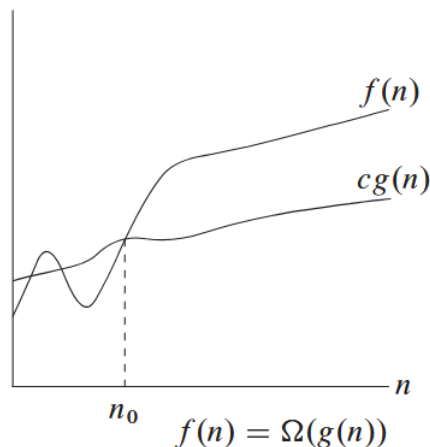
- **Multiplikation:**

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \implies f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

Die Komplexität des Produkts zweier Funktionen ist das Produkt der Komplexität der beiden Funktionen.

## 3.2 $\Omega$ Notation

Ähnlich zur O Notation, allerdings geht es hier um den *Best-Case* also minimale Anzahl der Schritten, die ein Algorithmus ausführt.



Wird auch wieder in  $\Omega$  und  $\omega$  aufgeteilt, die sich nur darin unterscheiden, wie strikt die Grenze ist.

### 3.2 (a) $\Omega$ Notation

Mathematische Definition:

$$\Omega(g(n)) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

Es existieren die positiven Konstanten  $c$  und  $n_0$ , sodass für alle  $n \geq n_0$  gilt, dass  $0 \leq c \cdot g(n) \leq f(n)$ . Das bedeutet, dass der Wachstumsfaktor von  $f(n) \geq c \cdot g(n)$  ist.

Die Berechnung von  $\Omega$  ist leider nicht immer so simpel wie die Berechnung von O Notation. Nehme zum Beispiel einen Linearen Suchalgorithmus, der eine Liste so lange durchläuft, bis er die gesuchte Zahl gefunden hat. Die Komplexität ist  $O(n)$ , da, wenn das Element an letzter Stelle steht alle Eingaben durchlaufen werden müssen. Gleichmaßen kann es aber sein, dass das Element an erster Stelle steht, was dann die Komplexität  $\Omega(1)$  besitzt. Dies muss allerdings durch Analyse des Algorithmus selbst erkannt werden und kann nicht aus der Funktionsrepräsentation ermittelt werden.

Gilt allerdings nicht sowas, wie vorzeitiger Abbruch bei Suche, so kann  $\Omega$  ähnlich zu  $O$  verwendet werden (Anhand vom Beispiel  $f(n) = 5n^2 + 2n$ ):

1. Finde den Term mit dem höchsten Wachstumsfaktor ( $5n^2$ )
2. Konstanten werden weggelassen ( $n^2$ )
3. Demnach ist  $f(n) = \Omega(n^2)$   
Da  $5n^2 + 2n$  für  $n \rightarrow \infty$  mindestens so schnell wächst wie  $n^2$ .

Um Werte für  $c$  und  $n_0$  zu finden, kann das Prinzip wie in O Notation verwendet werden, jedoch auf der Definition von  $\Omega$  angepasst (Umgekehrtes Gleichheitszeichen).

### 3.2 (b) $\omega$ Notation

Mathematische Definition:

$$\omega(g(n)) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)\}$$

Es existieren die positiven Konstanten  $c$  und  $n_0$ , sodass für alle  $n \geq n_0$  gilt, dass  $0 \leq c \cdot g(n) < f(n)$ .

Das bedeutet, dass der Wachstumsfaktor von  $f(n) > c \cdot g(n)$  ist.

Für die Bestimmung von  $\omega$  gilt das selbe wie für  $\Omega$ , nur das zusätzlich noch folgendes beachtet werden muss:

- Hat der Algorithmus einen konstanten Best-Case, so ist  $\omega$  nicht anwendbar, da  $\omega < 1$  sinnlos ist, da per Definition die Komplexität nicht kleiner als 1 sein kann und so der Best-Case schon durch  $\Omega$  definiert ist.
- Falls nicht konstant, dann muss bei  $\omega$  ähnlich zu Little-o herausgefunden werden, ob der Wachstumsfaktor von  $f(n)$  strikt größer ist als der Wachstumsfaktor der Funktion  $g(n)$ .

$$- \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

- Wenn  $\lim = \infty$ , so gilt  $\omega(g(n))$

- Andernfalls muss der Polynomgrad von  $g(n)$  verringert werden:  
 $\rightarrow n^x = n^{x-1} \implies x = 1$

### 3.2 (c) Rechenregeln

Sind sowohl für *Big* –  $\Omega$  als auch *Little* –  $\omega$  gültig

- **Konstanten:**

$$f(n) = a \text{ mit } a \in \mathbb{R}_{>0} \implies f(n) = \Omega(1)$$

Ist die Funktion konstant und positiv, so ist die Komplexität  $\Omega(1)$ .

- **Skalare Multiplikation:**

$$f(n) = \Omega(g(n)) \implies a \cdot f(n) = \Omega(g(n)) \text{ für } a > 0$$

Eine positive skalare Multiplikation der Funktion ändert die Komplexität nicht.

- **Addition:**

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \implies f_1(n) + f_2(n) = \Omega(\min\{g_1(n), g_2(n)\})$$

Die Komplexität der Summe zweier Funktionen ist der Minimalwert der Komplexität der beiden Funktionen.

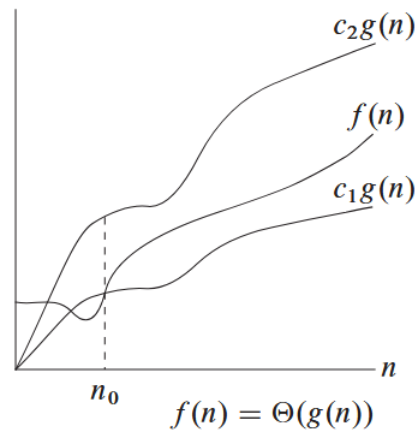
- **Multiplikation:**

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \implies f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

Die Komplexität des Produkts zweier Funktionen ist das Produkt der Komplexität der beiden Funktionen.

### 3.3 $\Theta$ Notation

$\Theta$  Notation kombiniert  $O$  und  $\Omega$  Notation. Das heißt sie stellt Durchschnittswachstum (Average-Case) einer Funktion dar und liegt somit zwischen  $O$  und  $\Omega$ .



Mathematische Notation:

$$\Theta(g(n)) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Es existieren die positiven Konstanten  $c_1, c_2$  und  $n_0$ , sodass für alle  $n \geq n_0$  gilt, dass  $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ .  
 $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$ .

Die Berechnung von  $\Theta$  läuft dementsprechend auch ähnlich zu  $O$  und  $\Omega$  ab (Anhand vom Beispiel  $f(n) = 5n^2 + 2n$ ).

1. Finde den Term mit dem höchsten Wachstumsfaktor ( $5n^2$ )
2. Konstanten werden weggelassen ( $n^2$ )
3. Demnach ist  $f(n) = \Theta(n^2)$   
Da  $5n^2 + 2n$  für  $n \rightarrow \infty$  mindestens so schnell wächst wie  $n^2$ .

Die Berechnung der Konstanten ist allerdings ein klein wenig komplizierter, da es eine mehr gibt. Prinzipiell bleibt es aber gleich:

- Simplifiziere die Gleichung:  $c_1 \cdot n^2 \leq 5n^2 + 2n \leq c_2 \cdot n^2 = c_1 \leq 5 + \frac{2}{n} \leq c_2$
- Da hier für alle  $n > 0$  der mittlere Term positiv ist, kann man  $n_0 = 1$  wählen.
- Dadurch erhalten wir  $c_1 \leq 5 + \frac{2}{1} = 7 \leq c_2$ , wodurch man hier die Konstanten dann z.B.  $c_1 = 7$  und  $c_2 = 7$  für  $n_0 = 1$  auswählen kann.

---

## 4 Sortieren

---

### 4.1 Sortierproblem

---

Sortieralgorithmen sind die wohl am häufigsten verwendeten Algorithmen. Hierbei wird als Eingabe eine Folge von Objekten gegeben, die nach einer bestimmten Eigenschaft sortiert werden. Der Algorithmus soll die Eingabe in der richtigen Reihenfolge (nach einer bestimmten Eigenschaft) zur Ausgabe umwandeln. Es wird hierbei meist von einer total geordneten Menge ausgegangen. (Alle Elemente sind miteinander vergleichbar).

Eine Totale Ordnung wird wie folgt definiert:

Eine Relation  $\leq$  auf  $M$  ist eine totale Ordnung, wenn:

- Reflexiv:  $\forall x \in M : x \leq x$   
(x steht in Relation zu x)
- Transitiv:  $\forall x, y, z \in M : x \leq y \wedge y \leq z \implies x \leq z$   
(Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch:  $\forall x, y \in M : x \leq y \wedge y \leq x \implies x = y$   
(Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass  $x = y$ )
- Totalität:  $\forall x, y \in M : x \leq y \vee y \leq x$   
(Alle Elemente müssen in einer Relation zueinander stehen)



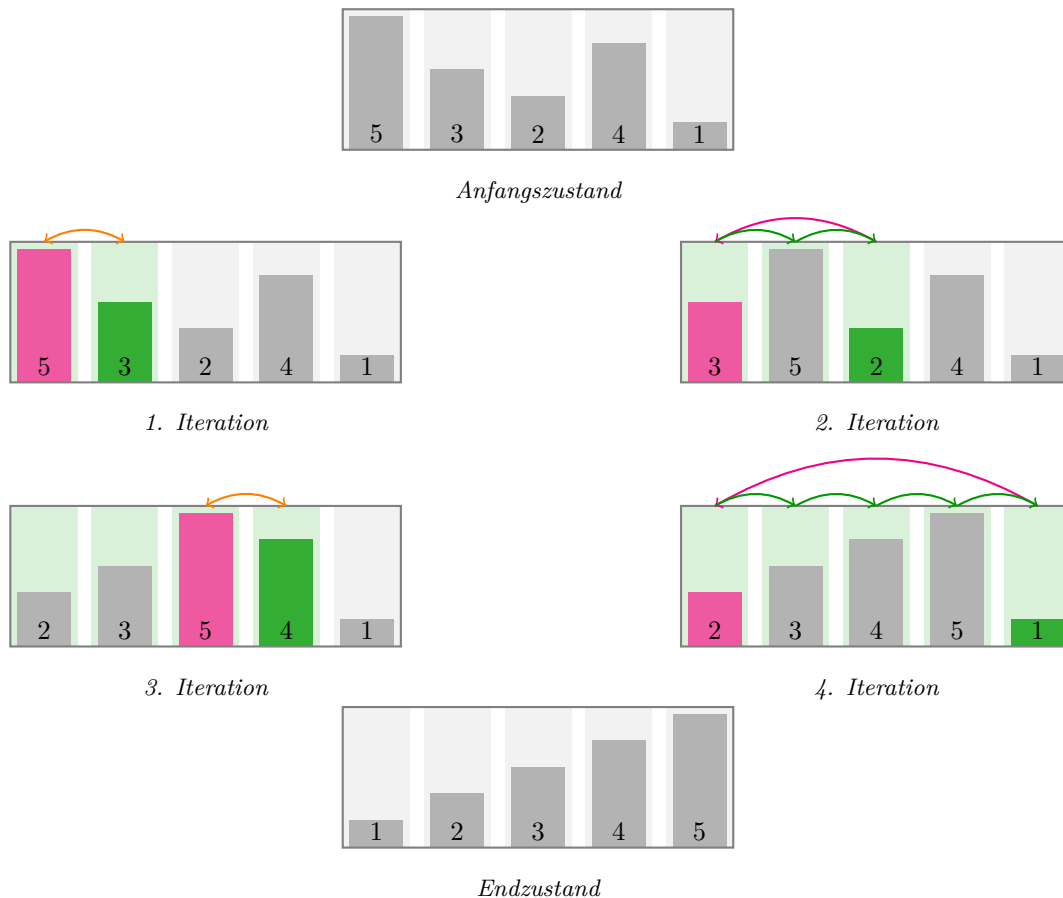
## 4.2 Insertion Sort

```
1 class InsertionSort {
2     void insertionSort(int[] arr) {
3         for (int i = 1; i < arr.length; i++) {
4             // 1 to n - 1
5             int key = arr[i];
6             int j = i - 1;
7             while (j >= 0 && arr[j] > key) {
8                 // Loops backwards through the array starting at i - 1
9                 // until it finds an element that is greater than the key or the beginning of the array
10                arr[j + 1] = arr[j];
11                // Shifts the element to the right
12                j--;
13            }
14            arr[j + 1] = key;
15            // Assigns the key to the correct position
16        }
17    }
18 }
```

### 4.2 (a) Vorgehensweise

Die Eingabe wird von links nach rechts durchlaufen startend bei  $i = 1$ . Das Element  $i$  wird dann mit allen Elementen verglichen, die links von  $i$  stehen, bis es 0 erreicht oder das die Einfügestelle gefunden wurde (Vor einem Element, das kleiner als das Element  $i$  ist). Die Elemente, die im betrachteten Bereich liegen und größer sind werden während dem Durchlauf eins nach rechts verschoben.

### 4.2 (b) Visuelle Darstellung



Grün ist das momentan betrachtete Element/Bereich. Magenta der Einfügestelle des Elements.

## 4.2 (c) Komplexität

- **Worst-Case:**

- Der Worst-Case ist ein array1, der in reverse order sortiert ist.
- Demnach muss jedes Element den kompletten array1 durchlaufen.
- Dies ergibt eine Worst-Case Laufzeit von  $\Theta(n^2)$

- **Best-Case:**

- Der Best-Case ist ein array1, der schon sortiert ist.
- Demnach muss kein Element verschoben werden, aber trotzdem muss bei jedem Element einmal geprüft werden, ob es größer als sein Vorgänger ist.
- Dies ergibt eine Best-Case Laufzeit von  $\Theta(n)$

- **Average-Case:**

- Der Average-Case ist ein array1, der in random order sortiert ist.
- Demnach muss für jedes Element der array1 durchschnittlich bis zur Hälfte durchlaufen werden.
- Nach der quadratischen Steigerung für große Zahlen ist die Hälfte aber irrelevant, weswegen  $\Theta(n^2)$  ist.

## 4.3 Bubble Sort

```
1 class BubbleSort {
2
3     void bubbleSort(int[] arr) {
4         for(int i = arr.length - 1; i > 0; i--) {
5             // Runs from arr.length - 1 to 0 (non exclusive)
6             // (i = 0 would immediately terminate)
7             boolean sorted = true;
8             for(int j = 0; j < i; j++) {
9                 // Runs from 0 to i - 1
10                if(arr[j] > arr[j + 1]) {
11                    // If the current element is greater than the next
12                    // Swap them
13                    int temp = arr[j];
14                    arr[j] = arr[j + 1];
15                    arr[j + 1] = temp;
16                    sorted = false;
17                }
18            }
19            if(sorted)
20                break;
21        }
22    }
23 }
```

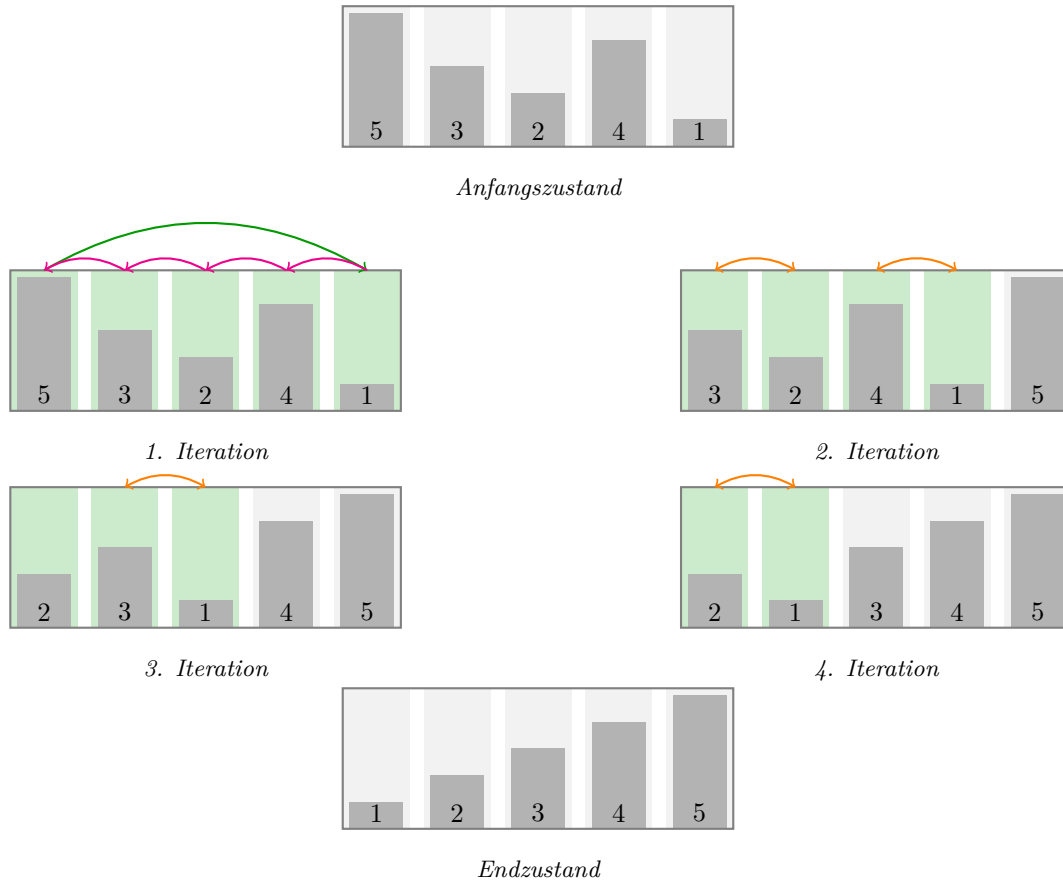
### 4.3 (a) Vorgehensweise

BubbleSort durchläuft die Eingabe umgekehrt zu InsertionSort: Während bei InsertionSort erst ein Element in einem Teil der Eingabe sortiert wird und der Bereich pro Iteration größer wird, wird bei BubbleSort zuerst der komplette array1 durchlaufen und beieinander liegende Elemente getauscht, wenn sie größer/kleiner sind und der Bereich mit Iteration weiter eingeschränkt. D.h., dass nach der ersten Iteration bereits das größte Element an richtiger Stelle steht, nach der zweiten das zweitgrößte etc.

Hier in dem Beispiel handelt es sich schon um einen optimierten BubbleSort. Dafür wird zusätzlich der Boolean sorted erstellt, der angibt, ob die Eingabe nach dem ersten durchlauf schon sortiert ist, was der Fall ist, wenn kein Element vertauscht wurde. Ist dies der Fall müssen keine weiteren Iteration mehr durchgeführt werden und der Algorithmus kann vorzeitig abgebrochen werden. Dies führt zu einem besseren Best-Case.

Im Vergleich zu InsertionSort ist BubbleSort meist ineffektiver als InsertionSort, obwohl sie die gleichen Komplexitäten haben. Das liegt daran, dass InsertionSort weniger Operationen ausführen muss.

### 4.3 (b) Visuelle Darstellung



Pfeile repräsentieren Bewegung über eine Iteration, nicht einzelne Schritte. Grün repräsentiert den bearbeiteten Bereich.

### 4.3 (c) Komplexität

- **Worst-Case:**

- Die Eingabe liegt in reverse order vor.
- Das heißt, das jedes Element immer vom Anfang bis zum Ende des Bereichs durchgewechselt werden muss.
- Die Komplexität beträgt also  $\Theta(n^2)$

- **Best-Case:**

- Die Eingabe ist bereits sortiert.
- Das heißt der Algorithmus muss die Eingabe nur einmal durchlaufen um zu schauen, ob Elemente getauscht werden.
- Die Komplexität beträgt also  $\Theta(n)$
- (Bei nicht optimierten BubbleSort, läuft der Algorithmus immer komplett durch  $\Rightarrow \Theta(n^2)$ )

- **Average-Case:**

- Die Eingabe ist zufällig sortiert.
- Im Durchschnitt müssen die Elemente dennoch in den meisten Fällen getauscht werden.
- Die Komplexität beträgt also  $\Theta(n^2)$

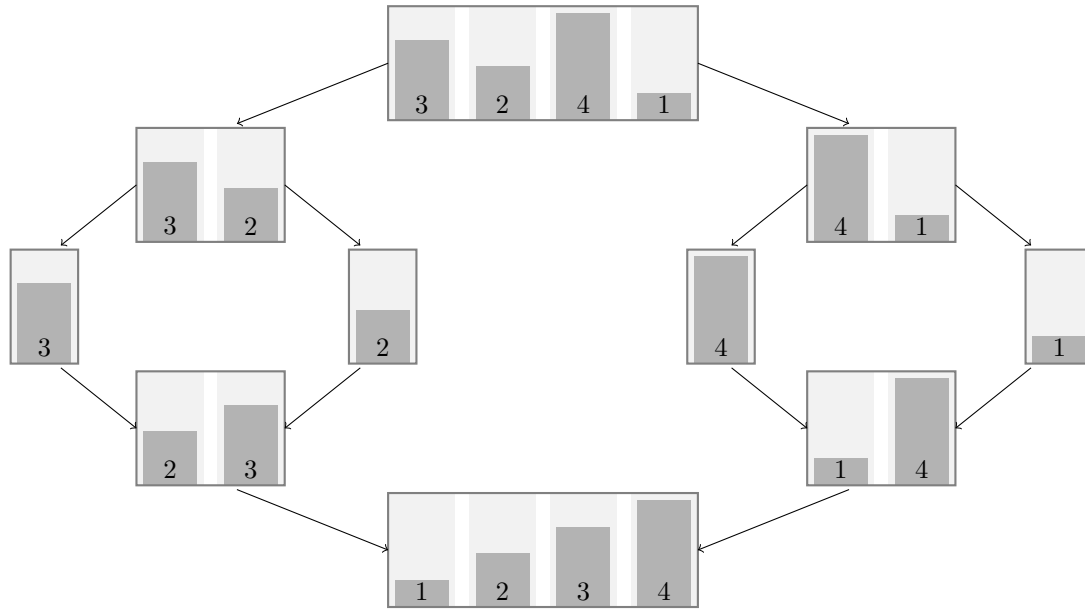
## 4.4 Merge Sort

```
1 class MergeSort {
2     void mergeSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // left < right, otherwise the region has no elements
5             int mid = (left + right) / 2; // Integer division -> round down
6             // Split the region into two halves and do the recursive calls
7             mergeSort(arr, left, mid);
8             mergeSort(arr, mid + 1, right);
9             // Merge the two (now sorted) halves
10            merge(arr, left, mid, right);
11        }
12    }
13
14    private void merge(int[] arr, int left, int mid, int right) {
15        int[] temp = new int[right - left + 1];
16        // Create a temporary array to store the merged elements
17
18        int p = left;
19        int q = mid + 1;
20        for (int i = 0; i < right - left + 1; i++) {
21            // Loops for each element in the region
22            if (q > right || (p <= mid && arr[p] <= arr[q])) {
23                // If p > mid the left half is finished, therefore the element needs to be in right half
24                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
25                temp[i] = arr[p];
26                p++;
27                // Adds the element at p to the temporary array and increases p
28            }
29            else {
30                temp[i] = arr[q];
31                q++;
32                // Adds the element at q to the temporary array and increases q
33            }
34        }
35        // Copy the merged elements from the temporary array back to the original array
36        for (int i = 0; i < right - left + 1; i++)
37            arr[left + i] = temp[i];
38        // left + 0 is the start of the region
39    }
40 }
```

### 4.4 (a) Vorgehensweise

Die Eingabe wird jeweils immer in der Mitte in zwei Teile aufgeteilt, die jeweils wieder aufgeteilt werden. Dies passiert so lange, bis alle Elemente einzeln vorhanden sind. Danach werden immer zwei dieser entstandenen Teillisten so zusammengeführt, dass sie geordnet sind. Dies wird dann wieder durchgeführt, bis alle Elemente in der Eingabe vorhanden sind und nun auch sortiert. Dieses Prinzip wird auch *Divide-and-Conquer* genannt. Bei *Divide* wird die Eingabe in zwei Teile aufgeteilt. Bei *Conquer* werden diese Teile sortiert. Dies geschieht durch die Zusammenführung von den einelementigen Teillisten, die trivial sortiert sind.

#### 4.4 (b) Visuelle Darstellung



#### 4.4 (c) Komplexität

- **Worst-Case:**

- Der Algorithmus funktioniert unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Worst-Case Eingabe.
- Die Eingabe kann  $\log n$  ( $\log_2 n$ ) mal in zwei aufgeteilt werden kann. Zusätzlich benötigt der Algorithmus zum Kombinieren von den Teillisten  $n$
- Es ergibt sich also die Komplexität von  $\Theta(n \log n)$

- **Best-Case:**

- Wie zuvor angesprochen, läuft der Algorithmus unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Best-Case Eingabe und der Best-Case ist gleich dem Worst-Case.
- Es ergibt sich also  $\Theta(n \log n)$

- **Average-Case:**

- Wie oben, für alle Fälle gleich, also  $\Theta(n \log n)$

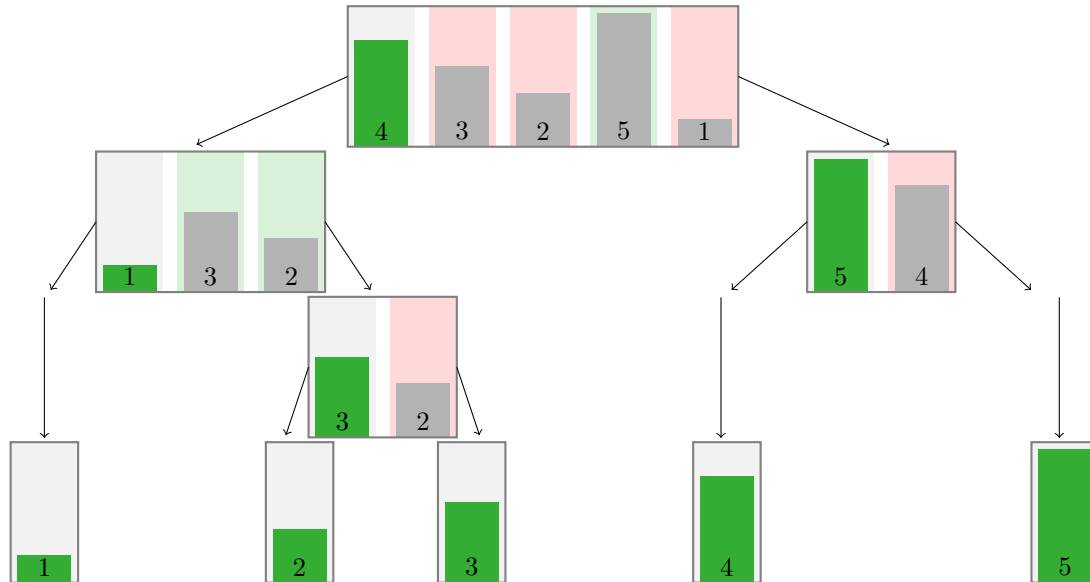
## 4.5 Quicksort

```
1 class Quicksort {
2     void quickSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // Region contains more than one element
5             int part = partition(arr, left, right);
6             quickSort(arr, left, part);
7             quickSort(arr, part + 1, right);
8         }
9     }
10
11     private int partition(int[] arr, int left, int right) {
12         int pivot = arr[left];
13         // Pivot is the first element in the region
14
15         int p = left - 1;
16         int q = right + 1;
17         while (p < q) {
18             do p++; while (arr[p] < pivot);
19             do q--; while (arr[q] > pivot);
20             // Increase / decrease p and q until the elements are bigger/smaller-equal pivot
21             if (p < q) {
22                 /* p < q here means that theres a number bigger equal pivot on the left side
23                  and a number smaller equal than the pivot on the right side
24                  Therefore, we swap them to sort them into their halves*/
25                 int temp = arr[p];
26                 arr[p] = arr[q];
27                 arr[q] = temp;
28                 // Swap arr[p] and arr[q]
29             }
30         } /* This loop runs until p and q cross each other
31            which means that */
32         return q;
33         // q is the index at which:
34         // all indices greater than q contain elements bigger equal pivot
35         // all indices smaller equal q contain elements smaller equal pivot
36     }
37 }
```

Quicksort funktioniert vom Prinzip ähnlich zu Mergesort. Auch hier wird die Eingabe in zwei Teillisten aufgeteilt und der rekursiv wiederholt. Hier findet die Sortierung allerdings anders statt. Anstatt die Sortierung durch die Zusammenführung zweier Listen zu realisieren, werden hier die einzelnen Elemente anhand des Vergleiches an einem anderen Elementes links oder rechts von diesem eingeordnet. Dies führt durch das *Divide-and-Conquer* Prinzip dazu, dass die Eingabe die Element in die zwei Teile, größer und kleiner des Pivots einordnet. Diese beiden Teile werden dann wiederum genauso behandelt, bis schließlich der gesamte array1 geordnet ist.

Bei der Implementation wird häufig anstatt den Pivot als erstes Element des Bereiches zu definieren, dieser zufällig gewählt, was zu einem besseren average-case führt, wenn die Eingabe bereits einigermaßen sortiert ist. Quicksort ist zwar in der Theorie in den meisten Situationen nicht unbedingt besser als Merge sort auf die Komplexität bezogen, in der Praxis aber oft schneller, durch die Ineffizienz von Kopieroperationen, die für Quicksort wegfallen.

#### 4.5 (a) Visuelle Darstellung



#### 4.5 (b) Komplexität

- **Worst-Case:**

- Im Worst-Case wird für pivot immer das größte oder kleinste Element verwendet, was sehr unausgeglichene Partitionen erzeugt.
- Dies würde eine Rekursionstiefe von  $n$  bedeuten
- Pro Rekursion muss dann der Bereich immernoch mit  $n$  durchlaufen werden
- Dies bedeutet eine Worst-Case Laufzeit von  $\Theta(n^2)$

- **Best-Case:**

- Im Best-Case wird immer das Element als pivot verwendet, das den Median der Liste bildet, was die Partitionen immer ausbalanciert.
- Dies bedeutet eine Rekursionstiefe von  $\log n$
- Pro Rekursion muss dann der Bereich immernoch mit  $n$  durchlaufen werden
- Dies bedeutet eine Best-Case Laufzeit von  $\Theta(n \log n)$

- **Average-Case:**

- Im Average-Case wird ein zufälliges Element als pivot verwendet, wodurch die Partitionen im Mittel gleich sind.
- Dies bedeutet eine Rekursionstiefe von  $\log n$
- Pro Rekursion muss dann der Bereich immernoch mit  $n$  durchlaufen werden
- Dies bedeutet eine Average-Case Laufzeit von  $\Theta(n \log n)$



## 4.6 Radix Sort

```
1 import java.util.ArrayList;
2
3 class RadixSort {
4     int D = 10; // possible unique digits
5     int d; // Max amount of digits
6     ArrayList<Integer>[] buckets = new ArrayList[D];
7
8     void radixSort(int[] arr) {
9         d = amountDigits(arr);
10        for (int i = 0; i < d; i++) {
11            // for each digit in the array, 0 least significant
12            for (int j = 0; j < arr.length; j++)
13                putBucket(arr, i, j);
14            // Sorts the numbers into their buckets
15            int a = 0;
16            for (int k = 0; k < D; k++) {
17                for (int b = 0; b < buckets[k].size(); b++) {
18                    arr[a] = buckets[k].get(b);
19                    a++;
20                }
21                buckets[k].clear();
22            }
23            // Reads out the buckets in order
24        }
25    }
26
27    private void putBucket(int[] arr, int i, int j) {
28        int z = arr[j] / (int) Math.pow(D, i) % D;
29        // Gets the ith digit of the number
30        int b = buckets[z].size();
31        // size is next free index
32        buckets[z].add(b, arr[j]);
33        // puts the number in the bucket z at position b
34        // Depending on implementation might need to increase size manually
35    }
36
37    private int amountDigits(int[] arr) {
38        int max = arr[0];
39        for (int i = 1; i < arr.length; i++) {
40            if (arr[i] > max)
41                max = arr[i];
42        }
43        // Get the biggest number
44        return (int) (Math.log(max)/Math.log(D) + 1);
45        // Get the amount of digits of the number
46        // log(max)/log10(D) is equal to log_D(max)
47    }
48 }
```

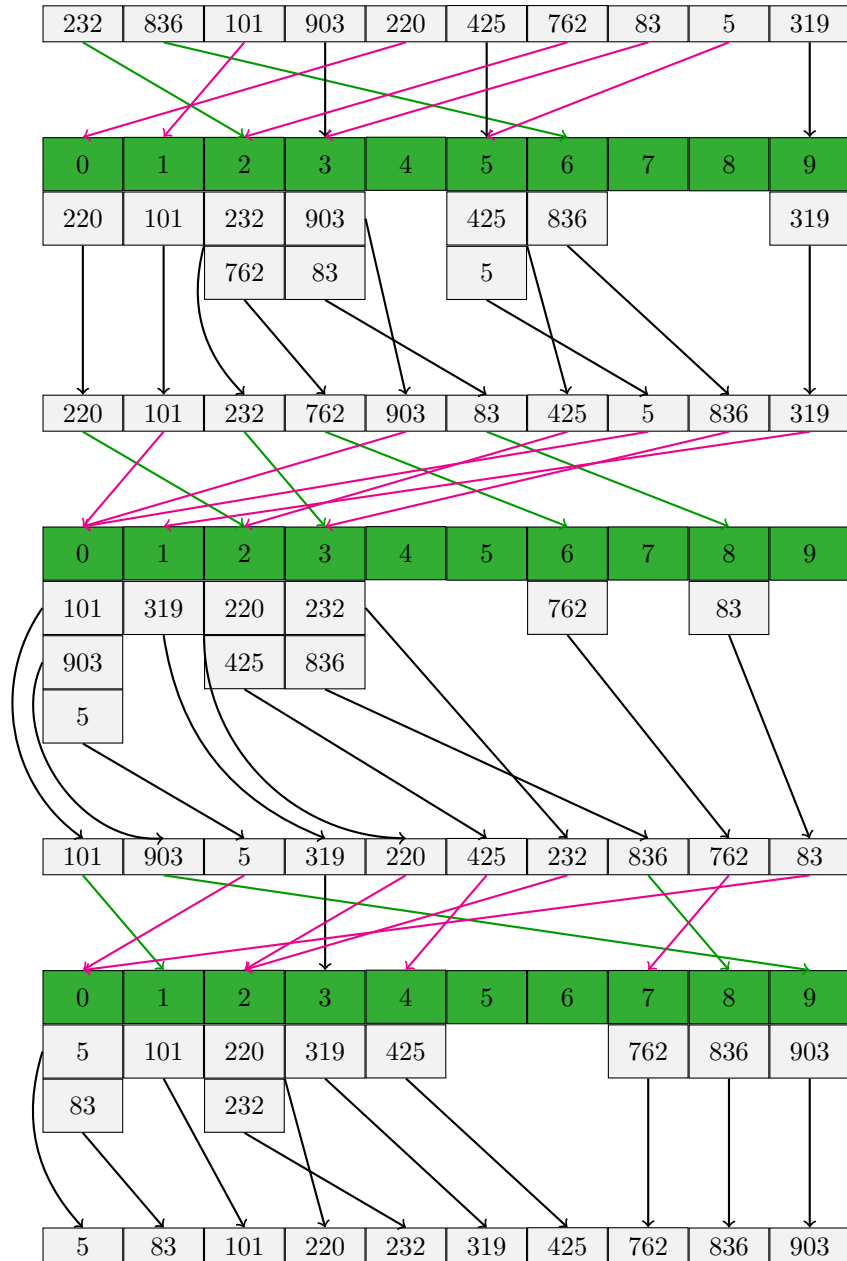
### 4.6 (a) Vorgehensweise

Bei RadixSort wird die Eingabe für jede Dezimalstelle sortiert. D.h., dass die Eingabe zuerst anhand von der 1er-Stelle sortiert wird, dann der 10er-Stelle, und so weiter.

Dies geschieht durch die Einordnung der Elemente in "Buckets", die jeweils einen möglichen Wert für die Dezimalstelle darstellen (z.B. {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}). Nachdem alle Werte in Buckets eingeordnet wurden, werden diese Buckets nun nach Signifikanz ausgelesen (0 ist kleiner als 9, also werden 0 zuerst ausgelesen) und nach der bearbeiteten Ziffer sortiert in die Eingabe zurückgefügt. Dadurch liegt der array1 für die Ziffer nun sortiert da.

Dies wird nun für die nächste Dezimalstelle wiederholt, wodurch die Eingabe jetzt für die ersten beiden Dezimalstellen sortiert ist. Dies wird wiederholt, bis alle Dezimalstellen durchlaufen sind, wodurch dann alle Werte sortiert sind.

#### 4.6 (b) Visuelle Darstellung



Die Farben haben keine spezielle Bedeutung und dienen nur der Visualisierung.

#### 4.6 (c) Komplexität

- Da bei RadixSort die Eingabe nur von der Anzahl der möglichen Ziffernvariationen  $D$ , der Eingabelänge  $n$  und die maximale Anzahl der Ziffern  $d$  abhängig ist, ist der Algorithmus für **Best-, Worst- und Average-case** gleich.
- Dieser beträgt im Allgemeinen  $O(d \cdot (n + D))$
- $D$  wird aber oft als Konstant angesehen, weshalb  $O(d \cdot n)$  oft verwendet wird.
- Wenn man zusätzlich noch  $d$  als konstant ansieht so ergibt sich lineare Laufzeit  $O(n)$
- Nähert sich  $D$   $n$  an, so ergibt sich allerdings eine Laufzeit von  $O(n \log n)$ , da  $d = \Theta(\log_D n)$  gilt.

## 5 Grundlegende Datenstrukturen

### 5.1 Stacks

Stacks operieren unter dem "First in - Last out" (FILO) Prinzip. Ähnlich zu einem Kartendeck, wo die unterste (Erste Karte) die ist, die als letztes gezogen wird.

Stacks werden normalerweise mit den folgenden Funktionen erstellt:

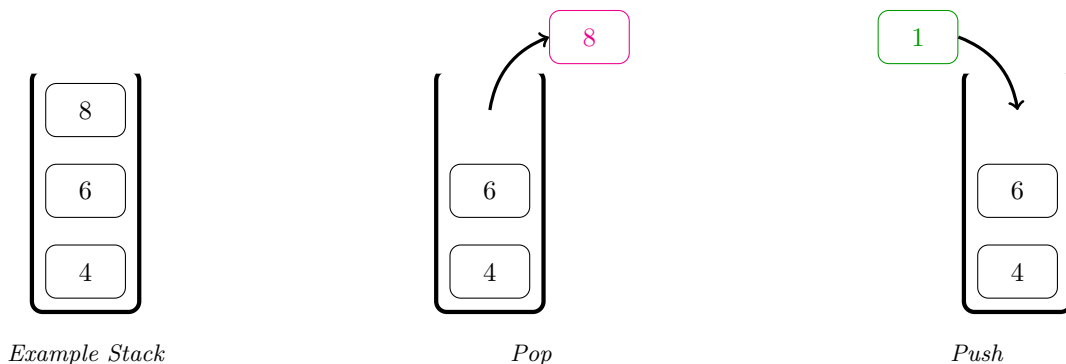
- `new(n)`: Erstellt einen neuen Stack.
- `isEmpty`: gibt an ob der Stack leer ist.
- `pop`: gibt das oberste Element des Stacks zurück und entfernt es vom Stack.
- `push(k)`: Fügt `k` auf den Stack hinzu

Eine mögliche Implementation auf Grundlage eines Arrays wäre:

```
1 class Stack {
2     private int[] arr;
3     private int top;
4     Stack(int size) {
5         arr = new int[size];
6         top = -1;
7         // Creates a new array with size
8     }
9     boolean isEmpty() {
10        return top < 0;
11        // Returns true if empty
12    }
13    int pop() { // O(1)
14        return arr[top--];
15        // Removes and returns the top element
16    }
17    void push(int k) { // O(1)
18        arr[++top] = k;
19        // Adds an element
20    }
21 }
```

Push und Pop schmeißen Fehlermeldung wenn Stack leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wäre es automatisch `IndexOutOfBoundsException`.

Oft werden Stacks auch mit variabler Größe implementiert. Dies kann über verschiedene Wege passieren, zum Beispiel Kopieren des arrays in einen größeren Array oder implementation über mehrere Arrays (z.B. über Linked List). Häufig wird das erstere so implementiert, dass der Array in einen Array mit doppelter Größe kopiert wird.



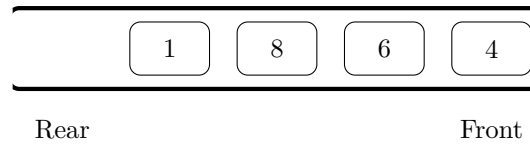
## 5.2 Queues

Queues funktionieren entgegengesetzt zu Stacks. Sie funktionieren nach dem FIFO-Prinzip (First in - First out). Kann als Warteschleife dargestellt werden. Die Person, die sich als erstes anstellt, kommt auch als erstes dran. Queues werden normalerweise mit den folgenden Funktionen erstellt:

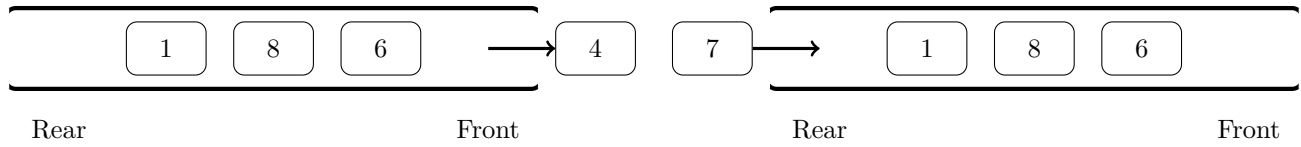
- `new(n)`: Erstellt einen neuen Queue.
- `isEmpty`: gibt an ob der Queue leer ist.
- `enqueue(k)`: Fügt `k` auf den Queue hinzu
- `dequeue`: gibt das erste Element des Queues zurück und entfernt es vom Queue.

Hier ist die Implementation für Queues wie folgt:

```
1 class Queue {
2     private int[] arr;
3     private int front;
4     private int back;
5
6     Queue(int size) {
7         arr = new int[size];
8         front = -1;
9         back = -1;
10    }
11
12    boolean isEmpty() {
13        return back == -1;
14    }
15
16    boolean isFull() {
17        return (front + 1) % arr.length == back;
18        // If front + 1 is equal to back, the queue is full
19        // Modulo makes this usable for cyclic arrays
20    }
21
22    void enqueue(int k) { // O(1)
23        if (isFull()) {
24            throw new RuntimeException("Queue is full");
25        } else {
26            if (isEmpty())
27                front = 0;
28            back = (back + 1) % arr.length;
29            // Modulo so that cyclic arrays work
30            arr[back] = k;
31        }
32    }
33
34    int dequeue() { // O(1)
35        if (isEmpty()) {
36            throw new RuntimeException("Queue is empty");
37        } else {
38            int temp = arr[front];
39            front = (front + 1) % arr.length;
40            // Modulo so that cyclic arrays work
41            if (front == back) {
42                front = -1;
43                back = -1;
44            }
45            // If front and back are equal, the queue is empty -> reset
46            return temp;
47        }
48    }
49 }
```

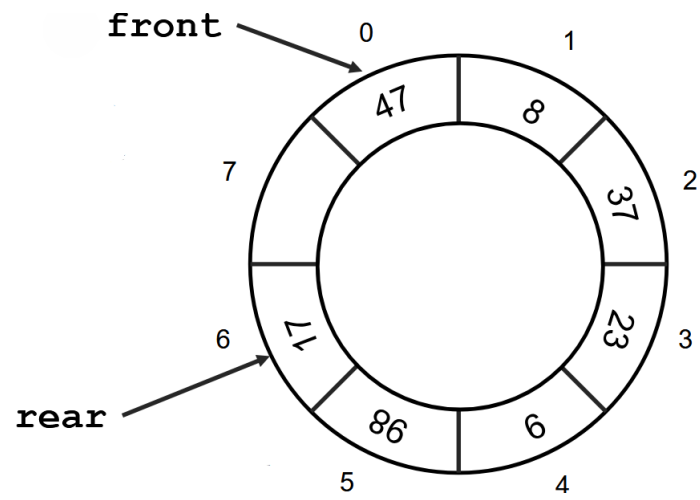


*Example non-cyclic Queue*



*Dequeue*

*Enqueue*



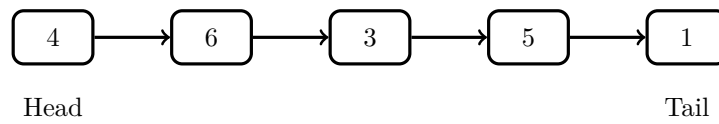
*Cyclic Queue*

## 5.3 Linked List

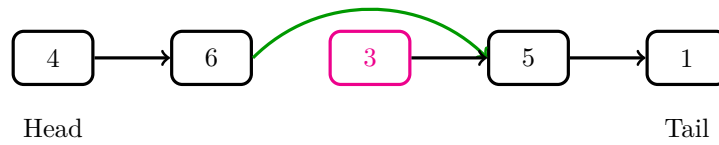
Eine einfache Linked List besteht aus mehreren Elementen, die jeweils immer einen Wert und eine Referenz auf das nächste Element in der Liste haben. Diese Struktur hat den Vorteil, dass sie keine festgelegte Größe hat, das Einfügen in  $O(1)$  stattfindet, einfach zu implementieren ist und im Speicher nicht als Block, sondern einzelne Referenzen steht. Eine einfache Linked List kann wie folgt implementiert werden:

```
1 class LinkedList {
2     class LinkedElement {
3         Integer key = null;
4         LinkedElement next = null;
5
6         LinkedElement(Integer key) {
7             this.key = key;
8         }
9     }
10    LinkedElement head = null; // First element in list
11    LinkedElement tail = null; // Last element in list
12
13    void insert(int k) { // O(1)
14        LinkedElement elem = new LinkedElement(k);
15        if (head == null) {
16            head = elem;
17            tail = elem;
18        }
19        else {
20            tail.next = elem;
21            tail = elem;
22        }
23    }
24
25    void delete(int k) { // O(n)
26        LinkedElement prev = null;
27        LinkedElement curr = head;
28        while (curr != null && curr.key != k) {
29            prev = curr;
30            curr = curr.next;
31        }
32        if (curr == null)
33            throw new UException("Element not found");
34
35        if (prev != null) {
36            prev.next = curr.next;
37            if (curr == tail)
38                tail = prev;
39        } else {
40            head = curr.next;
41        }
42    }
43
44    LinkedElement search(int k) { // O(n)
45        LinkedElement curr = head;
46        while (curr != null && curr.key != k)
47            curr = curr.next;
48        if (curr == null)
49            throw new UException("Element not found");
50        return curr;
51    }
52 }
```

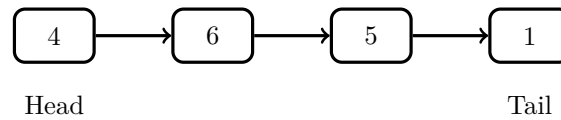
Diese Implementation benutzt einen Head und Tail, hat aber nur Referenz für das nächste Element in der Liste. Eine alternative Implementation wäre Tail wegzulassen und den Nodes eine previous-Referenz zu geben. Damit könnte man beim Einfügen das Element vorne an den Head anzuhängen und die neue Node als Head zuzuweisen. `search` bleibt gleich, bei `delete` muss lediglich die previous Referenz angepasst werden.



*Linked List*

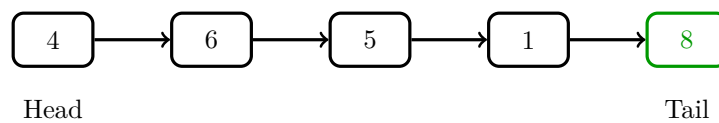


*Delete Outcome*



*Delete Quasi Outcome*

Die 3 Node wird zwar nicht wirklich "gelöscht", allerdings wird die Referenz aus der Liste genommen, wodurch keine Referenz mehr auf diese Node besteht.



*Insert of 8*

8 wird an tail angehängt und wird dann zum tail

## 5.4 Binary Search Tree

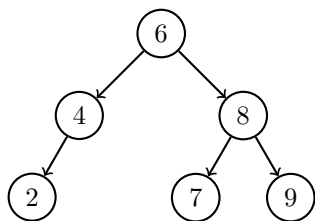
Ein Binary Search Tree ist eine Datenstruktur, die aus mehreren Nodes besteht, die jeweils pointer zu drei Nodes besitzt: Left, Right und Parent.

Hierbei repräsentiert Left und Right die Nodes, die unter der current Node stehen und Parent die, die über der current Node steht. Dabei ist im Binary Search Tree (Im Gegensatz zum normalen Search Tree) Left immer kleiner als die Node und Right immer größer gleich der Node.

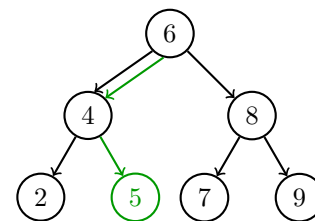
Dies erlaubt es Elemente in dem Tree schnell zu finden, da nicht alle Elemente durchlaufen werden müssen, sondern immer nur ein Pfad, bei dem das Element größer/kleiner ist.

Ein idealer Binary Search Tree ist so balanciert, dass beide Seiten des Baumes die selbe Anzahl an Knoten besitzen. Dies wäre eine ideale Höhe von  $h = \log n$ . Ein schlechter Binary Search Tree allerdings ist unbalanciert, so dass der Worst-Case so aussieht, dass alle Nodes jeweils maximal ein Kind haben. Dies wäre effektiv gleich einer LinkedList.

```
1 class BSTree {
2     class BSTNode {
3         Integer key;
4         BSTNode left;
5         BSTNode right;
6         BSTNode parent;
7         BSTNode(Integer k) {
8             key = k;
9         }
10    }
11    BSTNode root;
12    void insert(BSTNode z) { // Omega(1), O(h), Theta(h)
13        BSTNode x = root; // Traversal starting from the root
14        BSTNode px = null; // Parent of x, initially null
15        while(x != null) {
16            px = x;
17            if (z.key < x.key)
18                x = x.left;
19            else
20                x = x.right;
21        } // Traversing the tree until finding the insertion point
22        z.parent = px; // Sets the parent of the node to be inserted
23        if (px == null) // px only null if the tree is empty -> loop never runs -> z is root
24            root = z;
25        else if (z.key < px.key) // Key smaller -> left child
26            px.left = z;
27        else // Key bigger -> right child
28            px.right = z;
29        // May add the same node twice as it doesn't check for duplicates
30    }
```



Before insert



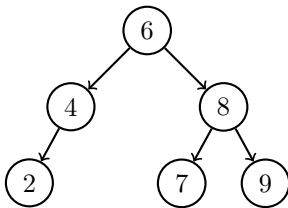
Insert 5



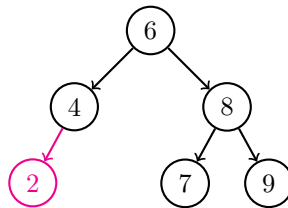
```

1  void delete(BSTNode z) { // Omega(1), O(h), Theta(log n)
2      if (z.left == null) // If z has no left, transplants the right to z's position
3          transplant(z, z.right);
4      else if (z.right == null) // If z has no right, transplants the left to z's position
5          transplant(z, z.left);
6      else { // If z has both left and right children
7          BSTNode y = z.right;
8          while (y.left != null)
9              // Finds the next biggest element of z = smallest in right subtree of z
10             y = y.left;
11          if (y.parent != z) { // If the next biggest element y is not child of z
12              transplant(y, y.right); // Transplants the right child of y to y's position
13              y.right = z.right; // The right child of y becomes the right child of z
14              y.right.parent = y; // The parent of the right child of y becomes y
15          }
16          transplant(z, y); // Transplants y to z's position
17          y.left = z.left; // The left child of y becomes the left child of z
18          y.left.parent = y; // The parent of the left child of y becomes y
19      }
20  }
21  void transplant(BSTNode u, BSTNode v) { // O(1)
22      // Transplants v to the parent of u
23      if (u.parent == null) // If u is the root, v becomes the new root
24          root = v;
25      else if (u == u.parent.left) // If u is a left child, v becomes a left child
26          u.parent.left = v;
27      else // If u is a right child, v becomes a right child
28          u.parent.right = v;
29      if (v != null) // If v is not null, v becomes a child of u's parent
30          v.parent = u.parent;
31  }

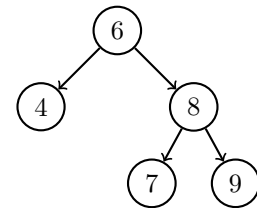
```



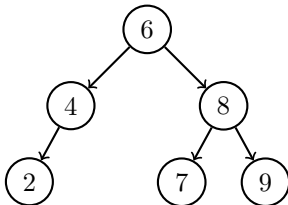
Leaf Deletion



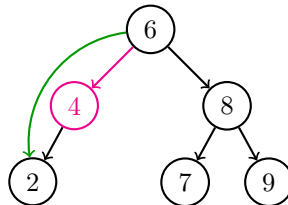
Delete 2



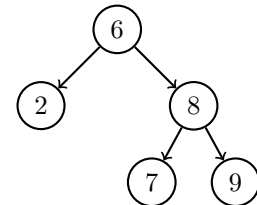
Result



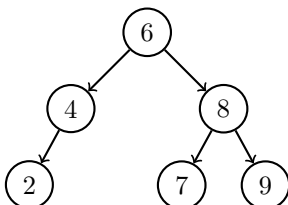
Half-Leaf Deletion



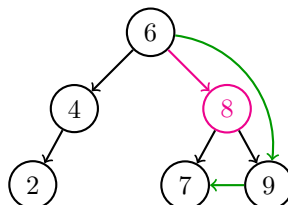
Delete 4



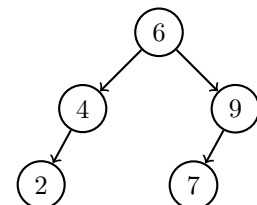
Result



Complete Node Deletion



Delete 8



Result

Leaves werden gelöscht, Half-Leaves durch Kind ersetzt, Complete Node durch Nachfolger (nächstgrößtes Element, kleinstes Element im rechten Teilbaum der Node) ersetzt.

```

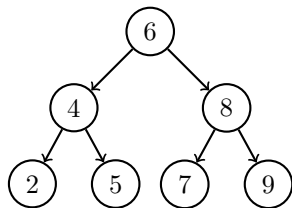
1  BSTNode iterativeSearch(int k) { // O(h), Omega(1), Theta(log n)
2      BSTNode curr = root;
3      while (curr != null && curr.key != k) {
4          if (k < curr.key)
5              curr = curr.left;
6          else
7              curr = curr.right;
8      }
9      return curr; // Returns null if element not found
10 }
11 BSTNode recursiveSearch(int k, BSTNode curr) { // O(h), Omega(1), Theta(log n)
12     if (curr == null)
13         return null;
14     if (k < curr.key)
15         return recursiveSearch(k, curr.left);
16     else if (k > curr.key)
17         return recursiveSearch(k, curr.right);
18     return curr; // Returns null if element not found
19 }

```

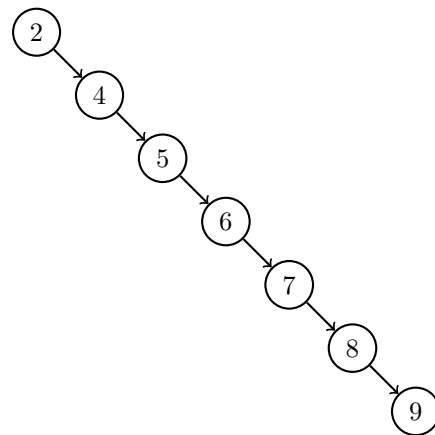
```

1  void traversal(BSTNode curr) { // O(n)
2      if (curr != null)
3          return;
4      // Any actions that should be done in a specific order can be done
5      // Here for preorder traversal
6      traversal(curr.left);
7      // Here for inorder traversal
8      traversal(curr.right);
9      // Here for postorder traversal
10     // Left and right can also be exchanged to traverse in reverse order
11 }
12 }

```



*Ideal balanzierter BST ( $h = \log n$ )*



*Worst-Case unbalanzierter BST ( $h = n$ )*

## 6 Fortgeschrittene Datenstrukturen

### 6.1 Red-Black Tree

Ein Red-Black Tree ist eine Art Binary-Search Tree. Zusätzlich zu diesem besitzen die Nodes in einem RB Tree noch das Attribut `color`. Die Nodes werden also entweder als `red` oder `black` definiert. Dies dient zur Einhaltung der Red-Black-Regeln, durch die die Effizienz der Datenstruktur im Vergleich zum BST verbessert wird.

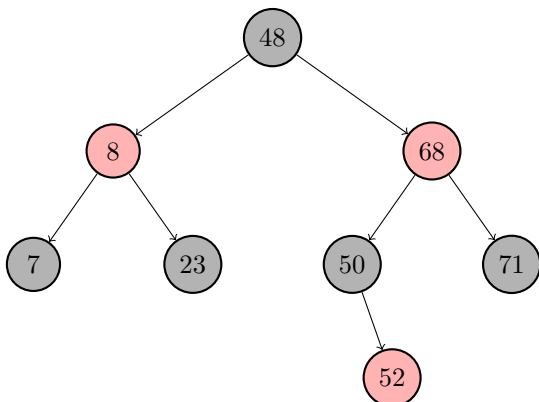
Die Regeln sind:

1. Jeder Knoten ist entweder schwarz oder rot
2. Die Wurzel ist schwarz
3. Rote Knoten haben keine Roten Kinder
4. Jeder Pfad von einem Knoten zu seinen Nachkommen besitzt die selbe Anzahl an schwarzen Knoten

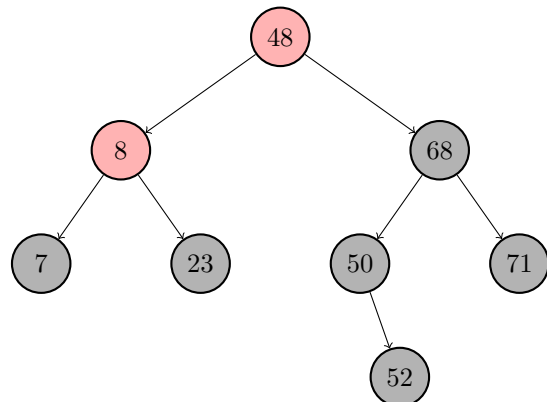
⇒ Hat ein Knoten nur ein Kind, so muss dieses Kind Rot sein, ansonsten ist die Anzahl an schwarzen Knoten auf dem Pfad unterschiedlich zu den anderen Pfaden.

Der Vorteil von RBT zu BST ist, dass während ein BST unausgewogen sein kann, was in einem Worst-Case von  $h = n$  resultiert, im RBT durch die Regeln eine maximale Höhe von  $h = \log n$  sichergestellt, was die Worst-Case Laufzeit der Algorithmen deutlich verbessert.

```
1 class RBTree {
2     class RBNode {
3         Integer key;
4         RBNode left;
5         RBNode right;
6         RBNode parent;
7         Color color;
8         RBNode(Integer k) {
9             key = k;
10        }
11    }
12    RBNode sent;
13    RBNode root = null;
14    RBTree() {
15        sent = new RBNode(null);
16        sent.color = Color.BLACK;
17        sent.left = sent;
18        sent.right = sent;
19        // Sentinel always points to itself ->
20        // node.parent.parent and its children will never result in null references
21    }
22    // Traversal and search are the same as BSTree
```



*Richtig Konstruierter RBT*



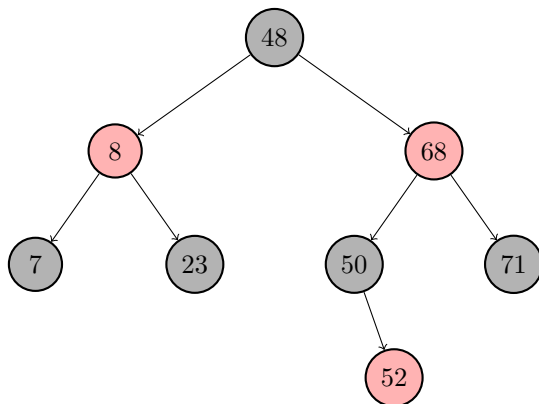
*Falsch Konstruierter RBT*

```

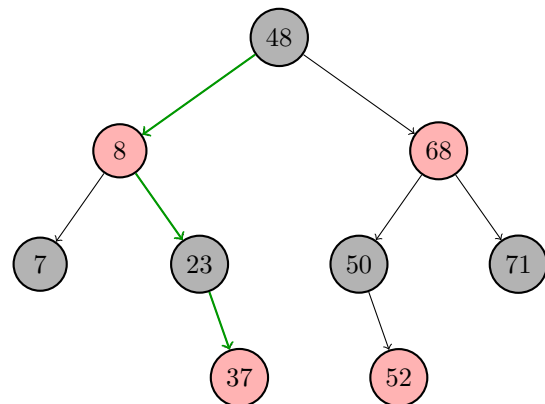
1  void insert(RBNode z) { // Omega(1), O(log n), Theta(log n)
2      // Very similar to BSTree, with addition of color and parent of sentinel instead of null
3      RBNode x = root; // Traversal starting from the root
4      RBNode px = sent; // Parent of x, initially sentinel unlike BST
5      while (x != null) {
6          px = x;
7          if (z.key < x.key)
8              x = x.left;
9          else
10             x = x.right;
11     } // Traversing the tree until finding the insertion point
12     z.parent = px; // Sets the parent of the node to be inserted
13     if (px == sent) // px only sentinel if the tree is empty -> loop never runs -> z is root
14         root = z;
15     else if (z.key < px.key) // Key smaller -> left child
16         px.left = z;
17     else // Key bigger -> right child
18         px.right = z;
19     z.color = Color.RED; // Sets color of new Node to red, will not necessarily stay red
20     fixColorsAfterInsertion(z); // Fixes colors in tree after insertion to maintain RB properties
21     // May add the same node twice as it doesn't check for duplicates
22 }

```

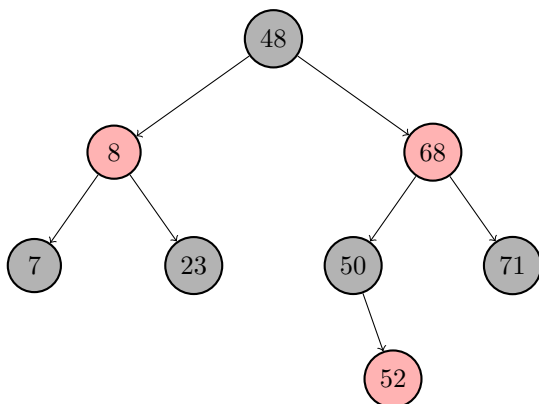
Einfügen funktioniert grundlegend gleich zu BST, allerdings wird am Ende die Farbe des neuen Knotens auf rot gesetzt und anschließend die Regeln des RBTs (falls verletzt) wieder hergestellt.



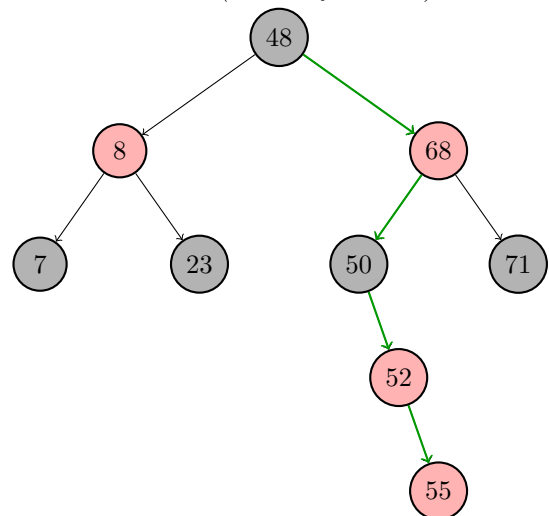
Before



Insert 37 (No colorfix needed)



Before

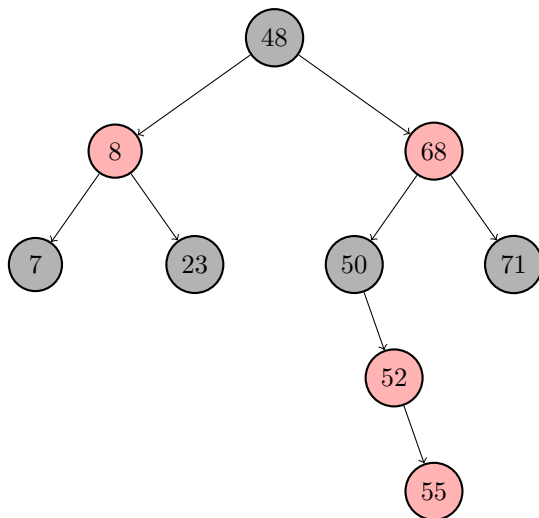


Insert 55 (Colorfix needed)

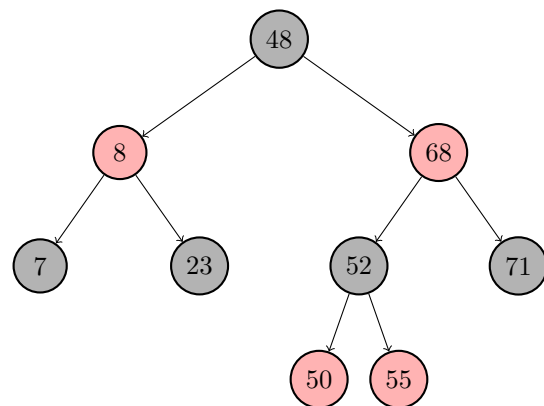
```

1  void fixColorsAfterInsertion(RBNode z) { // Omega(1), O(log n), Theta(log n)
2      while (z.parent.color == Color.RED) { // While z's parent is red
3          if (z.parent == z.parent.parent.left) { // If z's parent is a left child
4              RBNode y = z.parent.parent.right; // Gets sibling of z's parent
5              if (y != null && y.color == Color.RED) { // CASE 1: z's parent is a left child and
// sibling is red
6                  z.parent.color = Color.BLACK; // Set z's parent to black
7                  y.color = Color.BLACK; // Set z's uncle to black
8                  z.parent.parent.color = Color.RED; // Set z's grandparent to red
9                  z = z.parent.parent; // Set z to z's grandparent
10             } else { // CASE 2: z's parent is a left child and sibling is black
11                 if (z == z.parent.right) { // If z is a right child
12                     z = z.parent; // Set z to z's parent
13                     rotateLeft(z); // Rotate new z to left
14                 }
15                 z.parent.color = Color.BLACK; // Set z's parent to black
16                 z.parent.parent.color = Color.RED; // Set z's grandparent to red
17                 rotateRight(z.parent.parent); // Rotate z's grandparent to right
18             }
19         } else { // If z's parent is a right child
20             // Same as above but with right and left exchanged
21             RBNode y = z.parent.parent.left;
22             if (y != null && y.color == Color.RED) { // CASE 3: z's parent is a right child and
// sibling is red
23                 z.parent.color = Color.BLACK;
24                 y.color = Color.BLACK;
25                 z.parent.parent.color = Color.RED;
26                 z = z.parent.parent;
27             } else { // CASE 4: z's parent is a right child and sibling is black
28                 if (z == z.parent.left) {
29                     z = z.parent;
30                     rotateRight(z);
31                 }
32                 z.parent.color = Color.BLACK;
33                 z.parent.parent.color = Color.RED;
34                 rotateLeft(z.parent.parent);
35             }
36         }
37     }
38     root.color = Color.BLACK; // Set root to black, as it always should be
39 }

```



Fix needed at 55



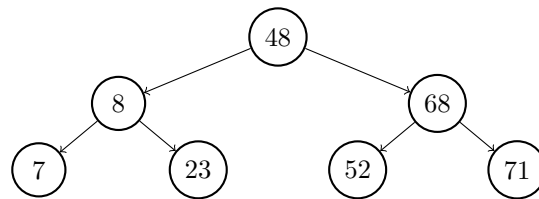
Fixed (Case 4)

Zum Wiederherstellen der RBT-Regeln muss der Baum an bestimmten Knoten rotiert werden.

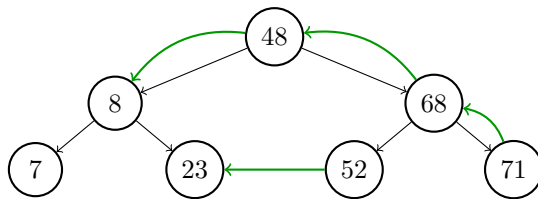
```

1  void rotateLeft(RBNode x) { // O(1)
2      RBNode y = x.right;
3      x.right = y.left; // Set x's right child to y's left child
4      if (y.left != null) // If y has a left child
5          y.left.parent = x; // Set y's left child's parent to x
6      y.parent = x.parent; // Set y's parent to x's parent
7      if (x.parent == sent) // If x is the root, set y to be the root
8          root = y;
9      else if (x == x.parent.left) // If x is a left child, set x's parent's left child to y
10         x.parent.left = y;
11     else // If x is a right child, set x's parent's right child to y
12         x.parent.right = y;
13     y.left = x; // Set y's left child to x
14     x.parent = y; // Set x's parent to y
15 }
16 void rotateRight(RBNode x) { // O(1)
17     // Same as rotateLeft but with right and left exchanged
18     RBNode y = x.left;
19     x.left = y.right;
20     if (y.right != null)
21         y.right.parent = x;
22     y.parent = x.parent;
23     if (x.parent == sent)
24         root = y;
25     else if (x == x.parent.right)
26         x.parent.right = y;
27     else
28         x.parent.left = y;
29     y.right = x;
30     x.parent = y;
31 }

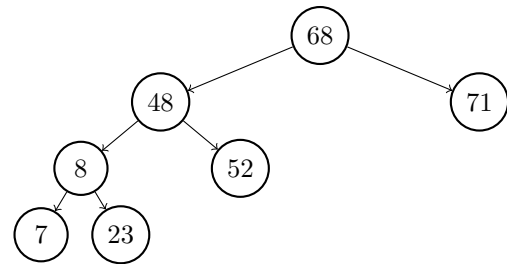
```



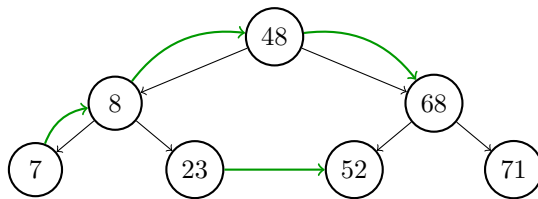
Before



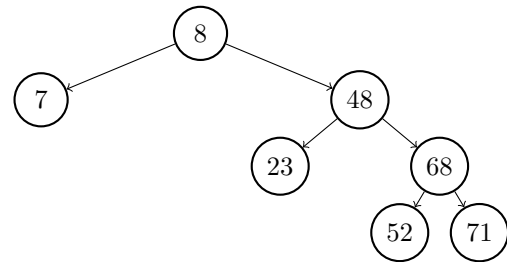
*rotateLeft(root)*



*rotateLeft(root) result*



*rotateRight(root)*

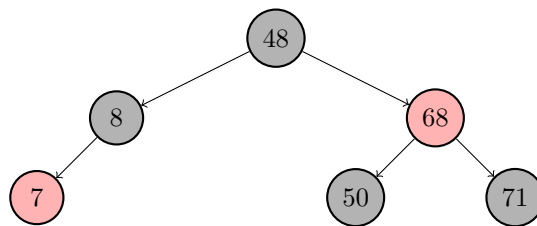


*rotateRight(root) result*

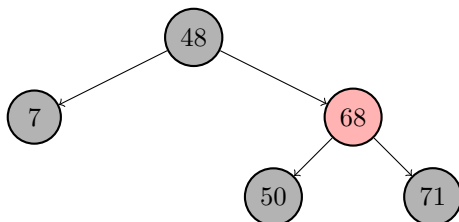
```

1  void delete(RBNode z) { // Omega(1), O(log n), Theta(log n)
2      RBNode a = z.parent; // a represent node with black depth imbalance
3      int dbh = 0; // delta black height, -1 for right, 1 for left leaning
4      if (z.left == null && z.right == null) { // CASE 1: z is a leaf
5          if (z.color == Color.BLACK && z != root) { // If z is black
6              if (z == z.parent.left) // If z is a left child
7                  dbh = -1; // Set delta black height to -1
8              else // If z is a right child
9                  dbh = 1; // Set delta black height to 1
10         }
11         transplant(z, null); // Transplant null to zs parent
12     } else if (z.left == null) { // CASE 2: z only has a right child
13         RBNode y = z.right;
14         transplant(z, z.right);
15         y.color = z.color;
16     } else if (z.right == null) { // CASE 3: z only has a left child
17         RBNode y = z.left;
18         transplant(z, z.left);
19         y.color = z.color;
20     } else { // CASE 4: z has two children
21         RBNode y = z.right;
22         a = y;
23         boolean wentLeft = false;
24         while (y.left != null) { // find next biggest Node
25             a = y;
26             y = y.left;
27             wentLeft = true;
28         }
29         if (y.parent != z) { // If next biggest element is not child of z
30             transplant(y, y.right);
31             y.right = z.right;
32             y.right.parent = y;
33         }
34         transplant(z, y);
35         y.left = z.left;
36         y.left.parent = y;
37         if (y.color == Color.BLACK) {
38             if (wentLeft) // Tree imbalanced depending on y location
39                 dbh = -1;
40             else
41                 dbh = 1;
42         }
43         y.color = z.color;
44     }
45     if (dbh != 0) // If black height imbalance
46         fixColorsAfterDeletion(a, dbh);
47 }

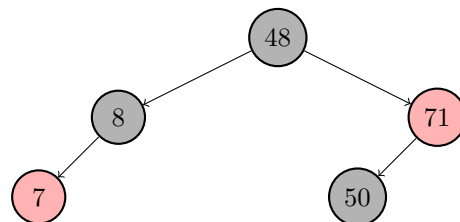
```



Before



Delete 8 (No colorfix needed, Case 3)



Delete 68 (Colorfix needed, Case 4, dbh = 1, skewed left)

```

1  void fixColorsAfterDeletion(RBNode a, int dbh) { // Omega(1), O(log n), Theta(log n)
2      if (dbh == -1) { // Extra black node on the right
3          RBNode x = a.left;
4          RBNode b = a.right;
5          RBNode c = b.left; // Left child of right child of a
6          RBNode d = b.right; // Right child of right child of a
7          if (x != null && x.color == Color.RED) {
8              // Easy case: x is red
9              x.color = Color.BLACK;
10         } else if (a.color == Color.BLACK
11             && b.color == Color.RED) {
12             // Case 1: a black, b red
13             rotateLeft(a);
14             a.color = Color.RED;
15             b.color = Color.BLACK;
16             fixColorsAfterDeletion(a, dbh);
17         } else if (a.color == Color.RED
18             && b.color == Color.BLACK
19             && (c == null || c.color == Color.BLACK)
20             && (d == null || d.color == Color.BLACK)) {
21             // Case 2a: a red, b black, c and d black
22             a.color = Color.BLACK;
23             b.color = Color.RED;
24         } else if (a.color == Color.BLACK
25             && b.color == Color.BLACK
26             && (c == null || c.color == Color.BLACK)
27             && (d == null || d.color == Color.BLACK)) {
28             // Case 2b: a black, b black, c and d black
29             b.color = Color.RED;
30             if (a == a.parent.left)
31                 dbh = 1;
32             else if (a == a.parent.right)
33                 dbh = -1;
34             else
35                 dbh = 0;
36             fixColorsAfterDeletion(a.parent, dbh);
37         } else if (b.color == Color.BLACK
38             && c != null && c.color == Color.RED
39             && (d == null || d.color == Color.BLACK)) {
40             // Case 3: a either, b black, c red, d black
41             rotateRight(b);
42             c.color = Color.BLACK;
43             fixColorsAfterDeletion(a, dbh);
44         } else if (b.color == Color.BLACK
45             && d != null && d.color == Color.RED) {
46             // Case 4: a either, b black, c either, d red
47             rotateLeft(a);
48             b.color = a.color;
49             a.color = Color.BLACK;
50             d.color = Color.BLACK;
51         }

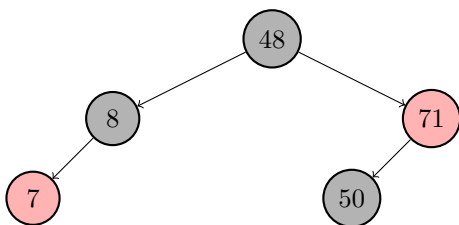
```



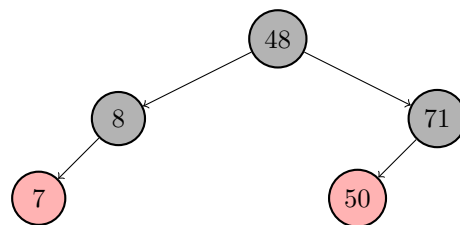
```

52     } else { // Extra black node on the left
53         // Same as above but with right and left exchanged
54         RBNode x = a.right;
55         RBNode b = a.left;
56         RBNode c = b.right; // Right child of left child of a
57         RBNode d = b.left; // Left child of left child of a
58         if (x != null && x.color == Color.RED) {
59             // Easy case: x is red
60             x.color = Color.BLACK;
61         } else if (a.color == Color.BLACK
62             && b.color == Color.RED) {
63             // Case 1: a black, b red
64             rotateRight(a);
65             a.color = Color.RED;
66             b.color = Color.BLACK;
67             fixColorsAfterDeletion(a, dbh);
68         } else if (a.color == Color.RED
69             && b.color == Color.BLACK
70             && (c == null || c.color == Color.BLACK)
71             && (d == null || d.color == Color.BLACK)) {
72             // Case 2a: a red, b black, c and d black
73             a.color = Color.BLACK;
74             b.color = Color.RED;
75         } else if (a.color == Color.BLACK
76             && b.color == Color.BLACK
77             && (c == null || c.color == Color.BLACK)
78             && (d == null || d.color == Color.BLACK)) {
79             // Case 2b: a black, b black, c and d black
80             b.color = Color.RED;
81             if (a == a.parent.right)
82                 dbh = 1;
83             else if (a == a.parent.left)
84                 dbh = -1;
85             else
86                 dbh = 0;
87             fixColorsAfterDeletion(a.parent, dbh);
88         } else if (b.color == Color.BLACK
89             && c != null && c.color == Color.RED
90             && (d == null || d.color == Color.BLACK)) {
91             // Case 3: a either, b black, c red, d black
92             rotateLeft(b);
93             c.color = Color.BLACK;
94             fixColorsAfterDeletion(a, dbh);
95         } else if (b.color == Color.BLACK
96             && d != null && d.color == Color.RED) {
97             // Case 4: a either, b black, c either, d red
98             rotateRight(a);
99             b.color = a.color;
100             a.color = Color.BLACK;
101             d.color = Color.BLACK;
102         }
103     }
104     // All cases except 2b mean end of the method in this or the next instance. 2b can go on tho.
105 }

```



Before fix



Fixed (Case 2a)

```

1  void transplant(RBNode u, RBNode v) { // O(1)
2      // Transplants v to the position of u
3      if (u.parent == sent) // If u is the root, v becomes the new root
4          root = v;
5      else if (u == u.parent.left) // If u is a left child, v becomes a left child
6          u.parent.left = v;
7      else // If u is a right child, v becomes a right child
8          u.parent.right = v;
9      if (v != null) // If v is not null, v becomes a child of u's parent
10         v.parent = u.parent;
11 }
12 }

```

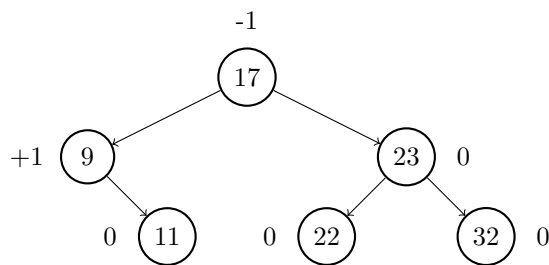
## 6.2 AVL Trees

Ein **Adelson-Velsky Landis Tree** ist ein BST, der sich selbst balanziert um eine Höhe von  $h = \log n$  zu garantieren. Er wird so definiert, **dass die Höhendifferenz von zwei Teilbäumen unter einem Knoten jeweils maximal 1 ist**.

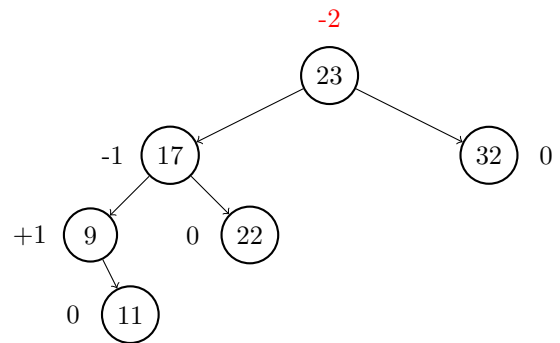
Im Vergleich zu RBT ist der AVL in der Höhe strikter. So ist die maximale Höhe im AVL  $h = 1.44 \cdot \log n$ , während er im RBT nur  $2 \cdot \log n$ . So haben AVLs zwar die bessere Effizienz in Suchvorgängen, jedoch benötigen sie beim Insert oder Delete meist mehr Rotationen. Demnach bieten sich AVLs eher bei Fällen an, wo mehr Suchvorgänge stattfinden im Vergleich zu den Insert/Delete Vorgängen. Muss der Baum oft modifiziert werden so bietet sich ein RBT besser an.

```
1 class AVLTree {
2     class AVLNode {
3         Integer key;
4         int height;
5         AVLNode left;
6         AVLNode right;
7         AVLNode(Integer k) {
8             key = k;
9             height = 1;
10        }
11    }
12    AVLNode root;
13    // Search and traversal like in BST
14    int height(AVLNode n) {
15        return (n == null) ? -1 : n.height;
16    }
17    void updateHeight(AVLNode n) {
18        n.height = 1 + Math.max(height(n.left), height(n.right));
19    }
20    int getBalance(AVLNode n) {
21        return (n == null) ? 0 : height(n.right) - height(n.left);
22    }
23 }
```

Visuell gleich zu einem Best-Case ausgewogenen BST.



*Balanced Tree*



*Unbalanced Tree*

```

1 // Rotations work a bit different as in other Trees as we don't have parents.
2 // Does not update the tree itself -> just returns the new root of the rotated subtree
3 // Does not need to check for null reference as right.left/left.right respectively
4 // are always well defined when called in insert and delete
5 AVLNode rotateLeft(AVLNode x) { // O(1)
6     AVLNode y = x.right;
7     AVLNode z = y.left;
8     y.left = x;
9     x.right = z;
10    updateHeight(x);
11    updateHeight(y);
12    return y;
13 }
14 AVLNode rotateRight(AVLNode x) { // O(1)
15     AVLNode y = x.left;
16     AVLNode z = y.right;
17     y.right = x;
18     x.left = z;
19     updateHeight(x);
20     updateHeight(y);
21     return y;
22 }

```

Während die Function ein wenig abgeändert ist, ist das Endergebnis bei richtiger Anwendung (nicht inPlace) gleich.

```

1  AVLNode insert(AVLNode partRoot, int key) { // O(h) = O(log(n))
2      if (partRoot == null) // Found insertion point
3          return new AVLNode(key);
4      else if (key < partRoot.key) // If node smaller than partRoot -> go left
5          partRoot.left = insert(partRoot.left, key);
6      else if (key > partRoot.key) // If node bigger than partRoot -> go right
7          partRoot.right = insert(partRoot.right, key);
8      else // If node == partRoot -> throw exception
9          throw new UException("Duplicate node");
10     return fixBalance(partRoot);
11     // Rebalance every node above the inserted node.
12 }

```

Funktioniert prinzipiell gleich zu den anderen Trees, aber ist nicht in-place und muss zusätzlich noch alle subtrees balanzieren.

```

1  AVLNode delete(AVLNode node, int key) { // O(h) = O(log(n))
2      if (node == null) { // Node not found
3          return node;
4      } else if (key < node.key) {
5          node.left = delete(node.left, key);
6      } else if (key > node.key) {
7          node.right = delete(node.right, key);
8      } else { // Node found -> Commence deletion
9          if (node.left == null || node.right == null) // If half leaf or leaf
10             node = (node.left == null) ? node.right : node.left;
11          else { // If complete node
12              AVLNode next = node.right;
13              while (next.left != null) {
14                  next = next.left;
15              }
16              node.key = next.key;
17              node.right = delete(node.right, next.key);
18          }
19      }
20      return fixBalance(node); // Rebalance every node above the deleted node
21  }

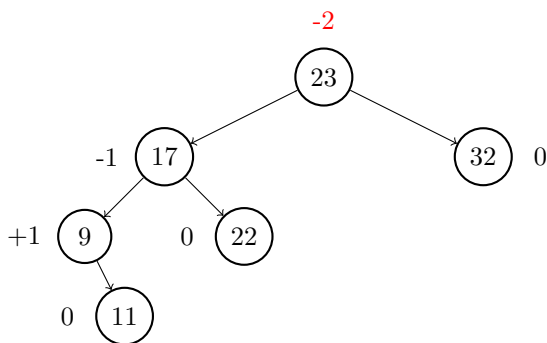
```

Wie bei rotate und insert, prinzipiell gleich zu den anderen Trees, aber ist nicht in-place und muss zusätzlich alle subtrees balanzieren.

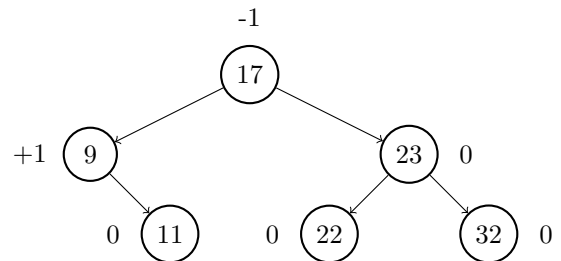
```

1  AVLNode fixBalance(AVLNode z) { // O(1)
2      updateHeight(z);
3      int balance = getBalance(z);
4      if (balance > 1) { // If right heavy
5          if (height(z.right.right) > height(z.right.left)) {
6              z = rotateLeft(z);
7          } else {
8              z.right = rotateRight(z.right);
9              z = rotateLeft(z);
10         }
11     } else if (balance < -1) { // If left heavy
12         if (height(z.left.left) > height(z.left.right)) {
13             z = rotateRight(z);
14         } else {
15             z.left = rotateLeft(z.left);
16             z = rotateRight(z);
17         }
18     }
19     return z;
20 }
21 }

```



*Unbalanced Tree after insert of 11*



*Balanced Tree after fixup*

## 6.3 Splay Trees

Splay Trees sind BSTs, die sich mit jedem Aufruf neu reorganisieren. Dies tun sie indem sie das betrachtete Element an die Wurzel verschieben. So ist der Splay Tree nicht unbedingt wie RBT und AVLTree gut balanziert, jedoch besonders effektiv, wenn einige Elemente öfters gesucht werden als andere. Da diese Bäume sich nicht wirklich selbst balanzieren, sind Splay Trees ungeeignet für Fälle wo alle Werte ungefähr gleich viel gesucht werden, da im Durchschnitt mehr Zeit gebraucht wird ein Wert zu finden und diesen an die Wurzel zu verschieben als in anderen Trees.

```
1 class SplayTree extends BSTree {
2     //Same Nodes as BSTree
3     void splay(BSTNode z) { // O(h)
4         while(z != root) {
5             if (z.parent.parent == null) // If father is root
6                 zig(z);
7             else {
8                 if (z == z.parent.parent.left.left || z == z.parent.parent.right.right)
9                     zigZig(z);
10                else
11                    zigZag(z);
12            }
13        }
14    }
15    void zig(BSTNode z) { // O(1)
16        if (z == z.parent.left)
17            rotateRight(z.parent);
18        else
19            rotateLeft(z.parent);
20    }
21    void zigZig(BSTNode z) { // O(1)
22        if (z == z.parent.left) {
23            rotateRight(z.parent.parent);
24            rotateRight(z.parent);
25        } else {
26            rotateLeft(z.parent.parent);
27            rotateLeft(z.parent);
28        }
29    }
30    void zigZag(BSTNode z) { // O(1)
31        if (z == z.parent.left) {
32            rotateRight(z.parent);
33            rotateLeft(z.parent.parent);
34        } else {
35            rotateLeft(z.parent);
36            rotateRight(z.parent.parent);
37        }
38    }
39    //Rotate works the same as in RBTree
```

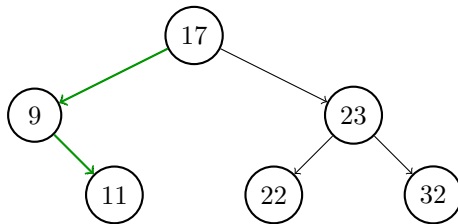
Sieht visuell praktisch wie ein normaler BST aus.



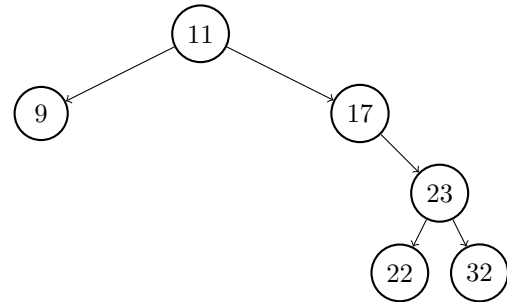
```

1  BSTNode search(int key) { // O(h), like BSTree, with additional splay
2      BSTNode x = root;
3      while(x != null && x.key != key) {
4          if (key < x.key)
5              x = x.left;
6          else
7              x = x.right;
8      }
9      if (x == null)
10         return null;
11     splay(x);
12     return root; // After splay the root is the searched node
13 }

```



*Search for 11*

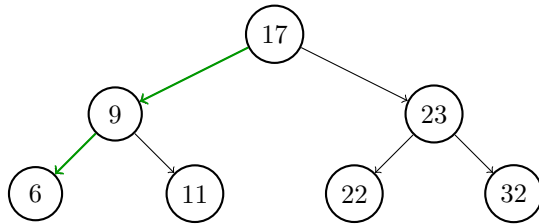


*Splay 11 (zigzag)*

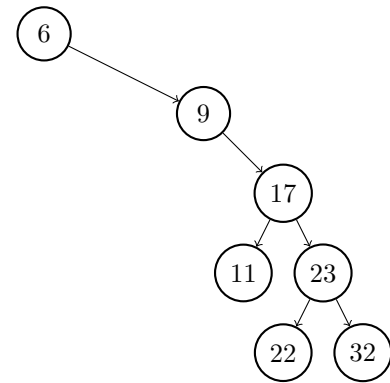
```

1  void insert (BSTNode z) { // O(h)
2      super.insert(z); // Inserts node using BSTrees insert
3      splay(z); // Splays node to the root
4  }

```



*Insert 6*

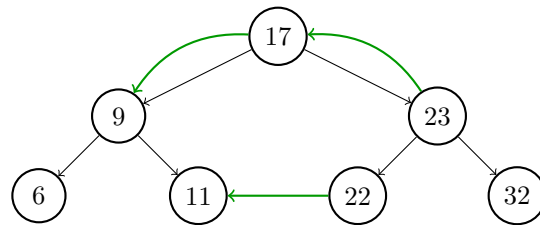


*Splay 6 (zigzig)*

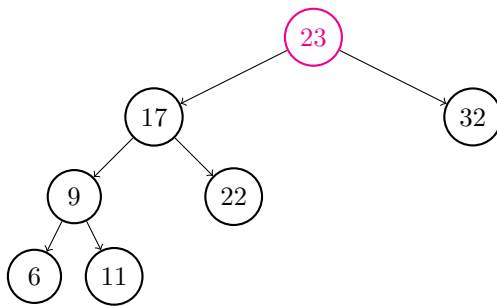
```

1  void delete (BSTNode z) { // O(h)
2      splay(z);
3      BSTNode r = root.right; // Save right child
4      BSTNode biggestL = root.left; // Save left child
5      root.right.parent = null; // Remove parent reference of right child of root
6      root.left.parent = null; // Remove parent reference of left child of root
7      root = biggestL; // Set root to left child
8      while (biggestL.right != null) // Get biggest node in left subtree
9          biggestL = biggestL.right;
10     splay(biggestL); // Splay biggest node -> becomes root
11     root.right = r; // Put right subtree back in place
12     r.parent = root; // Change parent of right subtree root
13 }
14 }

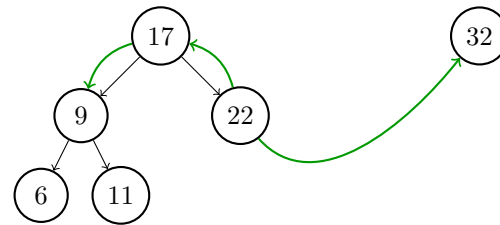
```



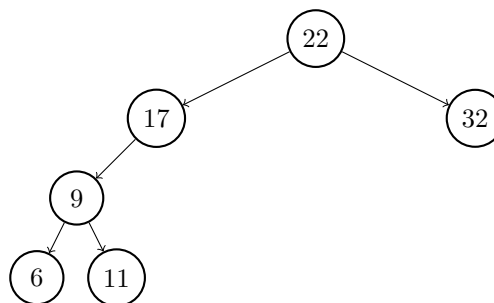
Before delete 23



Splay 23 (zig)



delete 23



Splay biggest node in left subtree (22) and append right subtree

## 6.4 Binary Heap Trees

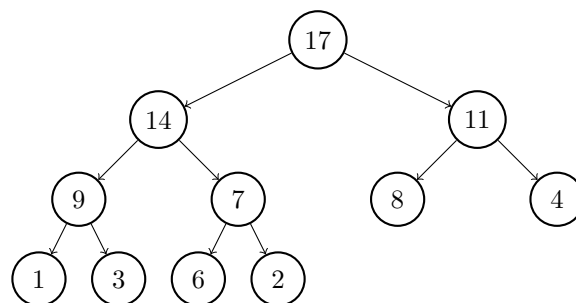
Im Allgemeinen ist ein Binary Heap wie folgt definiert:

- Bis auf das unterste Level ist der Baum vollständig gefüllt und im untersten Level ist er von links befüllt.
- $\forall x \neq \text{root} : x.\text{parent.key} \geq x.\text{key}$

Binary Heaps unterscheiden sich von den anderen hier behandelten Trees insofern, dass sie keine BSTs sind. Das heißt, dass rechte Kinder eines Knotens nicht unbedingt größer als dieser sind und linke Kinder dieses Knotens nicht unbedingt kleiner. Sie sind so organisiert, dass sie Werte nach Ebenen sortieren. Bei Max heaps zum Beispiel steht der größte Wert in der Wurzel und der kleinste Wert irgendwo in der untersten Ebene. So ergibt sich bei Max heaps also die Eigenschaft, dass die parent node einer node immer größer ist als die node selber. Dies erlaubt einen sehr schnellen Zugriff auf das größte Element. Diese Eigenschaft kann sehr gut genutzt werden um Werte zu sortieren. Zudem sind Binary Heaps anders konstruiert als andere Trees, sie besitzen nämlich keine Nodes perse, sondern sind nur über Positionen in einem array gespeichert. Die Beziehungen zwischen den Nodes ergeben sich durch Formeln:

- **Parent:**  $\text{parent}(i) = \lceil i/2 \rceil - 1$
- **Left Child:**  $\text{left}(i) = 2 \cdot (i + 1) - 1$
- **Right Child:**  $\text{right}(i) = 2 \cdot (i + 1)$

```
1 class BinaryMaxHeap {
2     Integer[] heap;
3     int size; // Size used to insert new elements at the first empty index
4     BinaryMaxHeap(int n) { // Creates new heap with size n
5         heap = new Integer[n];
6         size = 0;
7     }
8     BinaryMaxHeap(Integer[] arr) { // Creates heap from array
9         arrayToHeap(arr);
10    }
11    void arrayToHeap(Integer[] arr) { // Used to transfer array to heap
12        heap = arr.clone();
13        size = 0;
14        for (Integer i: arr) {
15            if (i == null) break;
16            size++;
17        }
18    }
19    int parent(int i) {
20        return (int) Math.ceil((double) i / 2) - 1;
21    }
22    int left(int i) {
23        return 2 * (i + 1) - 1;
24    }
25    int right(int i) {
26        return 2 * (i + 1);
27    }
28    boolean isEmpty() {
29        return size == 0;
30    }
31 }
```



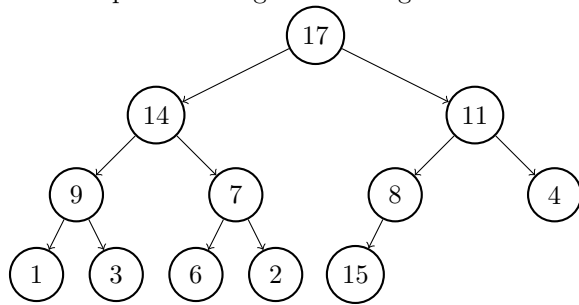
0	1	2	3	4	5	6	7	8	9	10	...
17	14	11	9	7	8	4	1	3	6	2	nil

```

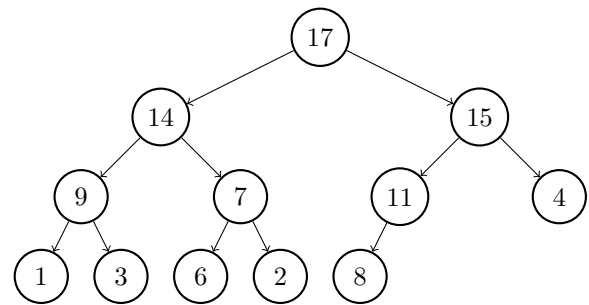
1  void insert(int k) { // O(h) = O(log n)
2      size++;
3      heap[size - 1] = k; // Add new element at first empty index
4      int i = size - 1;
5      while (i > 0 && heap[i] > heap[parent(i)]) { // Moves upward through tree
6          int temp = heap[i];
7          heap[i] = heap[parent(i)];
8          heap[parent(i)] = temp; // Swap elements if child is bigger than parent
9          i = parent(i); // Move up
10     }
11 }

```

Für min Heaps muss lediglich das Ungleichheitszeichen umgedreht werden.



*Insert 15, before fixup*



*After fixup*

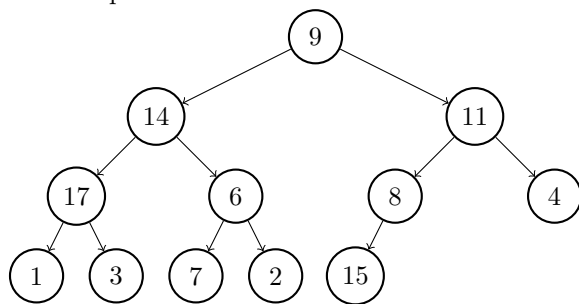
```

1  void heapify(int i) { // O(h) = O(log n)
2      // Used if heap is new unsorted array.
3      // If heap was build using insert, this should not be necessary
4      int max = i;
5      int l = left(i);
6      int r = right(i);
7      if(l < size && heap[i] < heap[l]) // If left child is bigger than i
8          max = l; // Set max to left
9      if(r < size && heap[max] < heap[r]) // if right child is bigger than max (i or left)
10         max = r; // Set max to right
11      if(max != i) { // If max is not i -> max is left or right
12          int temp = heap[i];
13          heap[i] = heap[max];
14          heap[max] = temp;
15          // Swap i and max
16          heapify(max); // max is now i
17          // Move down the tree
18      }
19  }

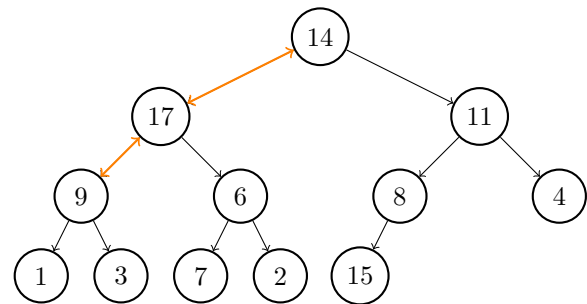
```

Tut essenziell das selbe wie der zweite Teil von Insert, nur in umgekehrter Reihenfolge (oben nach unten) und rekursiv.

Für Min heap muss wieder das Gleichheitszeichen umgedreht werden.



Before

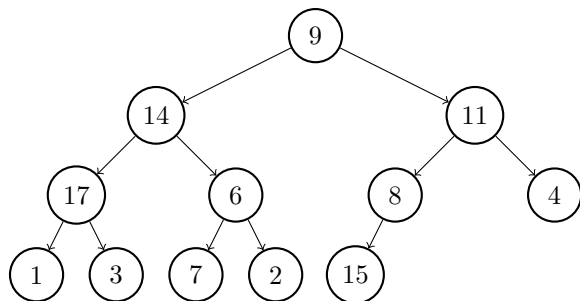


Heapify root

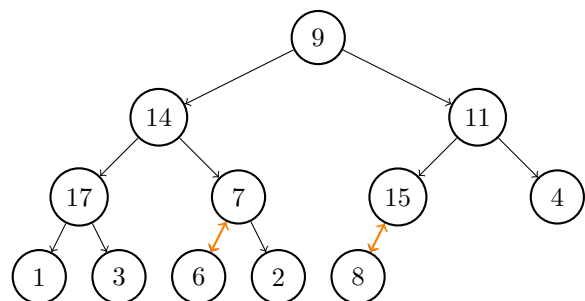
```

1  void buildHeap() { // O(nh) = O(n log n)
2      for(int i = parent(size - 1); i >= 0; i--)
3          heapify(i); // Calls heapify on each node starting from the second to last row
4          // Heap should now be sorted accordingly -> biggest node at root
5  }
6  int[] heapSort() { // O(nh) = O(n log n)
7      // Assumes new array for heap
8      buildHeap(); // Sorts heap accordingly
9      int[] res = new int[size]; // Create new array for sorted result
10     int i = 0;
11     while(!isEmpty())
12         res[i++] = extractMax(); // Extract max and add to array
13     return res; // Array is now sorted in reverse natural order
14 }
15 int extractMax() { // O(h) = O(log n)
16     if(isEmpty())
17         throw new UException("Underflow");
18     int max = heap[0]; // Biggest value at root
19     heap[0] = heap[size - 1]; // Sets root to last element
20     size--; // Decrease size
21     heapify(0); // Heapify new root
22     return max;
23 }
24 }

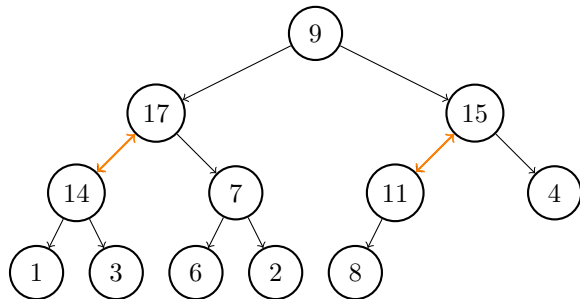
```



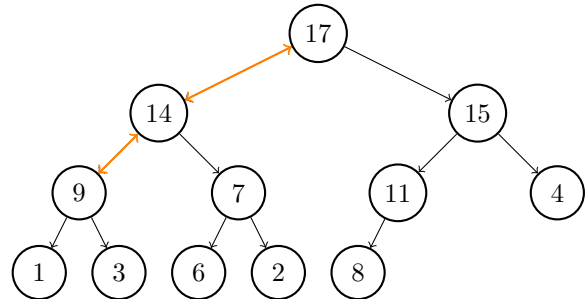
Before heapsort



heapify second to last level



heapify third to last level



heapify last level

0	1	2	3	4	5	6	7	8	9	10	11
17	15	14	11	9	8	7	6	4	3	2	1

Extracted array

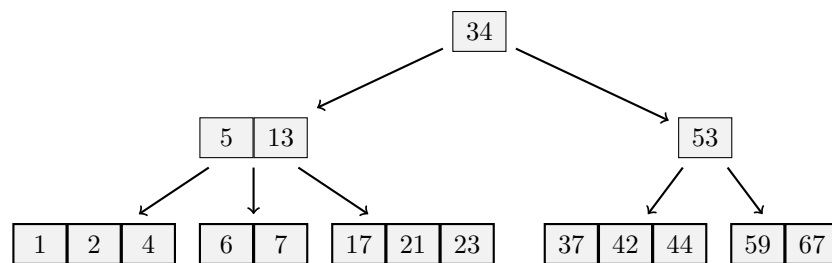
## 6.5 B-Tree

B-Trees (B hat keine festgelegte Bedeutung) unterscheiden sich sehr von den anderen Trees. Erstens sind sie keine Binary Trees. Ein B-Tree vom Grad  $t$  wird so definiert, dass

- Jede Node besitzt minimal  $t - 1$  Werte und  $t$  Kinder (Außer root)
- Jede Node besitzt maximal  $2 \cdot t - 1$  Werte und  $2 \cdot t$  Kinder
- Die Werte innerhalb eines Knotens sind aufsteigend sortiert
- Alle Blätter haben die gleiche Höhe
- Jeder innere Knoten mit  $m$  Werten hat  $m + 1$  Kinder  
 $\Rightarrow$  für alle Werte  $k_j$  in  $j$ -ten Kind gilt:  $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \dots \leq key[m - 1] \leq k_m$

Ein B-Tree kann somit in einem Knoten mehr als einen Wert und mehr als zwei Kinder besitzen. B-Trees balancieren sich zudem auch selber, wodurch sie sehr effektiv Operationen in logarithmischer Laufzeit ausführen. Zudem ist durch die Anzahl der Werte die in einem Knoten gespeichert werden können und die Anzahl an Kinder die ein Knoten haben kann die Höhe des Baumes deutlich niedriger. B-Trees bieten sich so sehr für Disk-Based Operationen an, da sie die Anzahl an Disk-Access reduziert im Vergleich zu anderen Trees. Sie bieten sich also besonders für sehr große Datenbanken auf Festplatten an, sind aber im Vergleich zu den anderen Trees bei kleineren Eingaben weniger effizient.

```
1 class BTree {
2     class BNode {
3         int[] keys; // array of all keys in the node
4         int t; // degree, defines the maximum number of keys in the node
5         BNode[] children; // array of children to the node
6         int n; // current number of keys in the node, used to find the first free index
7         boolean isLeaf; // true if the node is a leaf node -> no children
8     }
9     BNode root;
10    int t;
11    BTree(int t) {
12        root = null;
13        this.t = t;
14    }
15 }
```



*B-Tree example (Degree  $t = 2$ )*

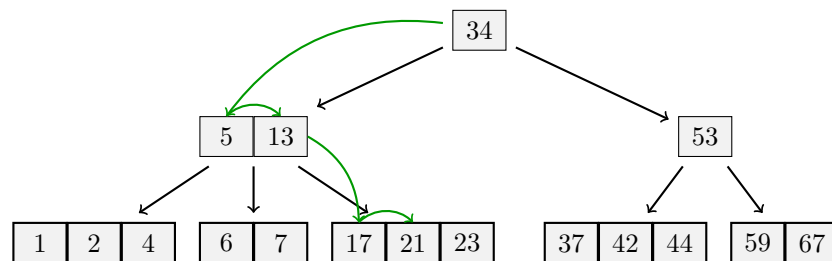


```

1 class BTree {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        BNode search(int k) {
12            int i = findKey(k); // find key
13            if (inNode(i)) // If k in node
14                return this;
15            if (isLeaf) // If k not in node and node is leaf, k doesnt exist in tree
16                return null;
17            return children[i].search(k); // search for k in corresponding child
18        }
19    }
20    void search (int k) {
21        if (root == null) {
22            System.out.println("Tree is empty");
23            return;
24        }
25        root.search(k); // Search for k
26    }
27 }

```

Der Suchalgorithmus ist relativ simpel. Er durchläuft die key-Werte der Wurzel, bis es beim Element angelangt, das größer oder gleich dem Suchwert ist. Ist der Wert gleich, so gibt es den Knoten zurück. Andernfalls, wenn der Knoten keine Kinder hat, existiert der Wert nicht, ansonsten durchsucht der Algorithmus das Kind, das den Wert beinhalten sollte rekursiv.



*Search 21*

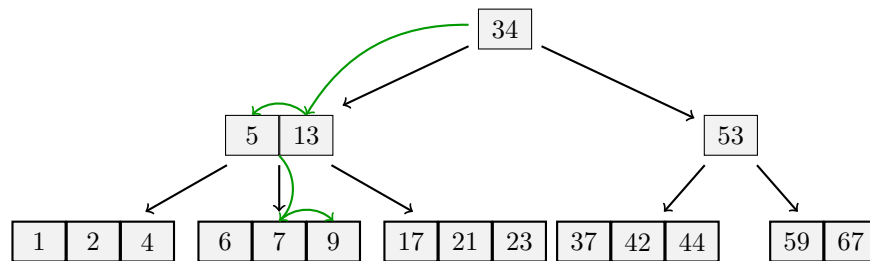
```

1 class BTree {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        void insertNonFull(int k) { // Omega(1), O(t), Theta(t)
12            int i = n - 1; // first free index
13            if (isLeaf) { // If node has no children
14                while (i >= 0 && keys[i] > k) {
15                    keys[i + 1] = keys[i];
16                    i--;
17                } // Moves through the array and moves elements bigger than k to the right
18                // Creates insertion point
19                keys[i + 1] = k; // Inserts k
20                n++; // Increases number of keys
21            } else { // If node has children
22                while (i >= 0 && keys[i] > k) i--; // Moves through array and finds insertion point
23                if (children[i + 1].n == 2 * t - 1) { // If insertion child is full
24                    splitChild(i + 1, children[i + 1]); // Splits insertion child
25                    if (keys[i + 1] < k) // If key at insertion point is smaller than k
26                        i++; // Move insertion point one to the right
27                }
28                children[i + 1].insertNonFull(k); // Insert k into insertion child
29            }
30        }
31        void splitChild(int i, BNode y) {
32            BNode z = new BNode(y.t, y.isLeaf); // Creates new node akin to y
33            z.n = t - 1; // Has half the number of keys as the node to be split
34            for (int j = 0; j < t - 1; j++) // Copies the second half of the keys to the new node
35                z.keys[j] = y.keys[j + t];
36            if (!y.isLeaf) { // If y has children
37                for (int j = 0; j < t; j++) // Copies the second half of the children to the new node
38                    z.children[j] = y.children[j + t];
39            }
40            y.n = t - 1; // Node now has half the keys it had before
41            for (int j = n; j > i; j--) // Searches for insertion point of new child
42                children[j + 1] = children[j];
43            children[i + 1] = z; // Inserts new child
44            for (int j = n - 1; j >= i; j--) // Creates insertion point for new key
45                keys[j + 1] = keys[j];
46            keys[i] = y.keys[t - 1]; // Inserts new key
47            n++; // Increases number of keys
48        }
49    }
50    void insert(int k) {
51        if (root == null) { // If tree is empty
52            root = new BNode(t, true); // Create root
53            root.keys[0] = k; // Insert k
54            root.n = 1; // Increase number of keys
55        } else { // If tree is not empty
56            if (root.n == 2 * t - 1) { // If root is full
57                BNode s = new BNode(t, false); // Create new node
58                s.children[0] = root; // set root as first child of new node
59                s.splitChild(0, root); // Split root
60                int i = 0;
61                if (s.keys[0] < k) i++; // If only key in new node is smaller than k, move insertion
point
62                s.children[i].insertNonFull(k); // Insert k in corresponding child
63                root = s; // sets new node as root
64            } else // If root is not full
65                root.insertNonFull(k); // Simply insert
66        }
67    }
68 }

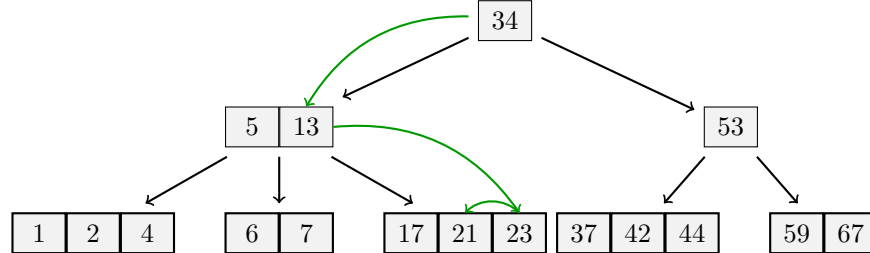
```

Prinzipiell folgt dieser Algorithmus einfach der Folge:

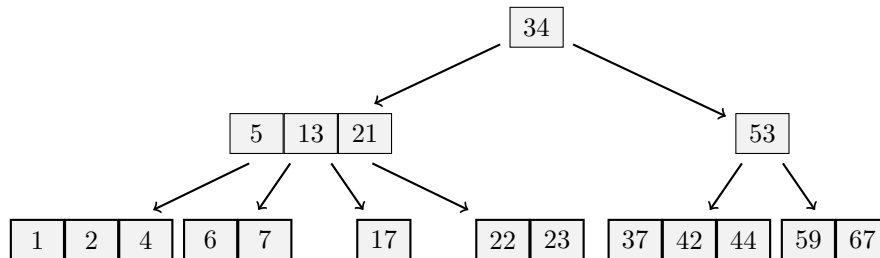
1. Finde Einfügapunkt
2. Wenn Node schon  $2 \cdot t - 1$  Werte besitzt, splitte es
  - Teile Node in zwei Nodes mit je  $t - 1$  Werten
  - Der mittlere Knoten wird in den Elternknoten eingefügt
  - Wenn dadurch der Elternknoten  $2 \cdot t - 1$  Werte besitzt, splitte diesen rekursiv nach oben
3. Wert am Einfügapunkt einfügen



*Insert 9 (Simple Case)*



*Before Insert 22 (Needs splitting)*



*Insert 22*

```

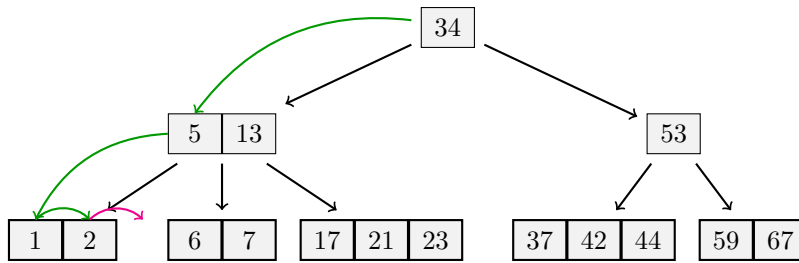
1 class BTree {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        int getPredecessor(int i) {
12            BNode child = children[i];
13            while (!child.isLeaf)
14                child = child.children[child.n];
15            return child.keys[child.n - 1];
16        }
17        int getSuccessor(int i) {
18            BNode child = children[i + 1];
19            while (!child.isLeaf)
20                child = child.children[0];
21            return child.keys[0];
22        }
23        void fill(int i) {
24            if (i != 0 && children[i - 1].n >= t) // If the previous child is filled more than half
25                borrowFromPrev(i);
26            else if (i != n && children[i + 1].n >= t) // If the next child is filled more than half
27                borrowFromNext(i);
28            else // If both children are not filled more than half
29                if (i != n) // Merge with next child
30                    merge(i);
31                else // Merge with previous child
32                    merge(i - 1);
33        }
34        void borrowFromPrev(int i) {
35            BNode child = children[i];
36            BNode sibling = children[i - 1]; // Get previous child, to be borrowed from
37            for (int j = child.n - 1; j >= 0; j--) // Move keys to the right
38                child.keys[j + 1] = child.keys[j];
39
40            if (!child.isLeaf)
41                for (int j = child.n; j >= 0; j--) // Move children to the right
42                    child.children[j + 1] = child.children[j];
43            child.keys[0] = keys[i - 1]; // Move key from parent to child
44            if (!child.isLeaf) // Move children if not child not a leaf
45                child.children[0] = sibling.children[sibling.n];
46            keys[i - 1] = sibling.keys[sibling.n - 1]; // Move key from sibling to parent
47            child.n++; // Increase number of keys in child
48            sibling.n--; // Decrease number of keys in sibling
49        }
50        void borrowFromNext(int i) {
51            BNode child = children[i];
52            BNode sibling = children[i + 1]; // Get next child, to be borrowed from
53            child.keys[child.n] = keys[i]; // Move key from parent to child
54            if (!child.isLeaf) // if child isnt a leaf move last first child of sibling to child
55                child.children[child.n + 1] = sibling.children[0];
56            keys[i] = sibling.keys[0]; // Move key from sibling to parent
57            for (int j = 1; j < sibling.n; j++) // Move keys to the left
58                sibling.keys[j - 1] = sibling.keys[j];
59            if (!sibling.isLeaf)
60                for (int j = 1; j <= sibling.n; j++) // Move children to the left
61                    sibling.children[j - 1] = sibling.children[j];
62            child.n++; // Increase number of keys in child
63            sibling.n--; // Decrease number of keys in sibling
64        }
65        void merge(int i) {
66            BNode child = children[i];
67            BNode sibling = children[i + 1];
68            child.keys[t - 1] = keys[i]; // Move key from parent to child
69            for (int j = 0; j < sibling.n; j++) // Move keys from sibling to second half of child
70                child.keys[j + t] = sibling.keys[j];

```

```

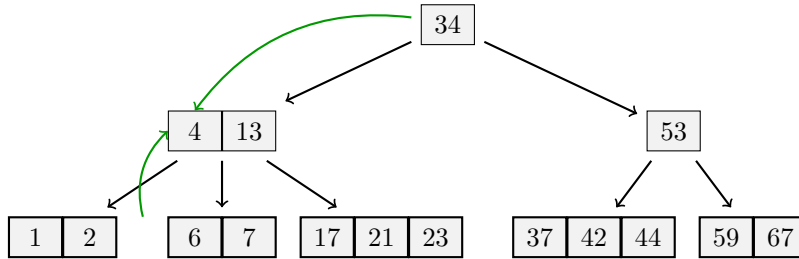
71         if (!child.isLeaf)
72             for (int j = 0; j <= sibling.n; j++) // Move children from sibling to second half of
child
73                 child.children[j + t] = sibling.children[j];
74         for (int j = i + 1; j < n; j++) // Move keys to fill the gap created by moving to child
75             keys[j - 1] = keys[j];
76         for (int j = i + 2; j <= n; j--) // Move children to fill the gap created by moving to child
77             children[j - 1] = children[j];
78         child.n += sibling.n + 1;
79         n--;
80     }
81     void delete(int k) {
82         int i = findKey(k);
83         if (inNode(i)) {
84             if (isLeaf)
85                 deleteFromLeaf(i);
86             else
87                 deleteFromNonLeaf(i);
88         } else {
89             if (isLeaf) {
90                 System.out.println("Key not found");
91                 return;
92             }
93             boolean flag = (i == n); // if key is present in subtree of last child
94             if (children[i].n < t) // If child that contains key has less than t keys
95                 fill(i); // Fill that child
96             if (flag && i > n) // If key is present in subtree of last child
97                 children[i - 1].delete(k); // Delete from that subtree
98             else // If key is not present in subtree of last child
99                 children[i].delete(k); // Delete from that subtree
100         }
101     }
102     void deleteFromLeaf (int i) {
103         for (int j = i + 1; j < n; j++) // Move all keys after i to the left
104             keys[j - 1] = keys[j];
105         n--; // reduce number of keys
106     }
107     void deleteFromNonLeaf (int i) {
108         int k = keys[i];
109         if (children[i].n >= t) { // If child that contains key has more than t keys
110             int pred = getPredecessor(i); // Get predecessor
111             keys[i] = pred; // Replace key with predecessor
112             children[i].delete(pred); // Delete predecessor from child
113         } else if (children[i + 1].n >= t) { // If child that contains key has more than t keys
114             int succ = getSuccessor(i); // Get successor
115             keys[i] = succ; // Replace key with successor
116             children[i + 1].delete(succ); // Delete successor from child
117         } else { // If both children have less than t keys
118             merge(i); // Merge children
119             children[i].delete(k); // Delete key from child
120         }
121     }
122 }
123 void delete(int k) {
124     if (root == null) {
125         System.out.println("Tree is empty");
126         return;
127     }
128     root.delete(k); // Delete k
129     if (root.n == 0) { // If root is empty
130         if (root.isLeaf) // If root is leaf -> tree is empty
131             root = null; // Delete root
132         else
133             root = root.children[0]; // Replace root with its child
134     }
135 }
136 }

```



#### *Delete 4 (Simple Case)*

Wenn der Knoten in einem Blatt steht kann er einfach rausgenommen werden.



#### *Delete 5 (Replace with predecessor in child)*

Im Fall, dass der Wert in einem inneren Knoten steht, geht der Algorithmus so vor:

1. Ersetze den Wert mit dem Vorgänger im Kind (Wenn Kind mindestens  $t$  Werte besitzt).
2. Wenn es nicht genug Werte besitzt, ersetze den Wert mit dem Nachfolger im Kind (Wenn Kind mindestens  $t$  Werte besitzt).
3. Wenn beide Kinder weniger als  $t$  Werte besitzen, verbinde die beiden Knoten und den zu löschenden Wert zu einem Knoten und lösche nun den Wert aus dem Knoten

Da dieser Algorithmus bereits beim Suchen den Baum umorganisiert um unnötige Operationen zu sparen gilt zudem, dass beim Suchen, wenn die Wurzel des Unterbaums, der den Wert beinhalten muss  $t - 1$  Werte besitzt und ein unmittelbares Geschwisternode mit mindestens  $t$  Werten besitzt, man einen Wert vom Parent in den Knoten tut und diesen mit dem Wert aus dem Geschwisternode ersetzt.

Wenn beide Nodes, also die Wurzel des Unterbaumes und die Geschwisternode  $t - 1$  Werte haben, verbinde sie.



---

## 7 Probabilistische Datenstrukturen

---



---

## 8 Graphen Algorithmen

---