AuD - Zusammenfassung

Moritz Gerhardt

1 Inhaltsverzeichnis

2 Was ist ein Algorithmus?

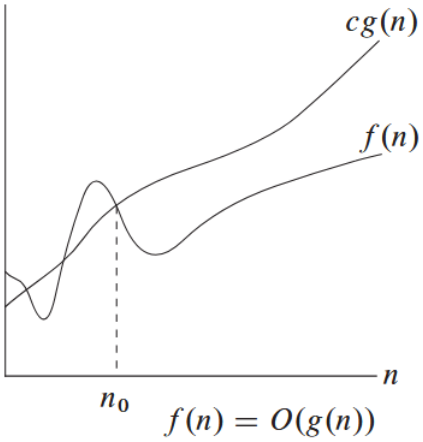
Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe.
Dabei sollte ein Algorithmus im allgemeinen folgende Vorraussetzungen erfüllen:

1. Bestimmt:
   - Determiniert: Bei gleicher Eingabe liefert der Algortihmus gleiche Ausgabe.
     $\Longrightarrow$ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
   - Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
     $\Longrightarrow$ 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. Andwendbar:
   - 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 verwerdendet. 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 $n \to \infty$ 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.
   - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{5n^2+2n}{n^2} = \lim_{n\to\infty} 5 + \frac{2}{n} = 5$
     $\impliedby \frac{2}{n}$ für $n \to \infty = 0$
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 $n \to \infty$ 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:**

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

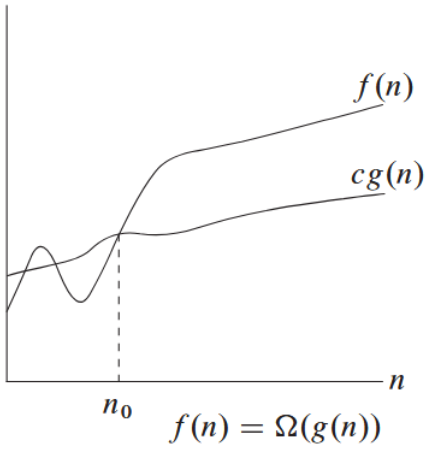
  Die Komplexität der Summe zweier Funktionen ist der Maximalwert der Komplexität der beiden Funktionen.
- **Multiplikation:**

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

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

3.2 Ω 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 Ω und ω aufgeteilt, die sich nur darin unterscheiden, wie strikt die Grenze ist.

3.2 (a) Ω 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 Ω 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. Gleichermaßen 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 Ω 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 $n \to \infty$ 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 Ω angepasst (Umgekehrtes Gleichheitsszeichen).

3.2 (b) ω 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 ω gilt das selbe wie für Ω, nur das zusätzlich noch folgendes beachtet werden muss:

- Hat der Algorithmus einen konstanten Best-Case, so ist ω 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 Ω definiert ist.
- Falls nicht konstant, dann muss bei ω ähnlich zu Little-o herausgefunden werden, ob der Wachstumsfaktor von $f(n)$ strikt größer ist als der Wachstumsfaktor der Funktion $g(n)$.
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
  - Wenn $lim = \infty$, so gilt $\omega(g(n))$
  - Andernfalls muss der Polynomgrad von $g(n)$ verringert werden:
    $\longrightarrow n^x = n^{x-1} \implies n = 1$

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:**

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

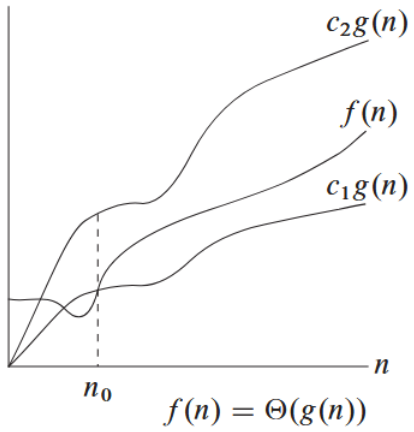
  Die Komplexität der Summe zweier Funktionen ist der Minimalwert der Komplexität der beiden Funktionen.
- **Multiplikation:**

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

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

3.3 Θ Notation

Θ Notation kombiniert O und Ω Notation. Das heißt sie stellt Durchschnittswachstum (Average-Case) einer Funktion dar und liegt somit zwischen O und Ω.



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 Θ läuft dementsprechend auch ähnlich zu O und Ω 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 $n \to \infty$ mindestens so schnell wächst wie n^2.

Die Berechnung der Konstanten ist allerding 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: $\forall x, y, z \in M : x \leq y \land y \leq z \implies x \leq z$
  (Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \land y \leq x \implies x = y$
  (Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass x = y)
- Totalität: $\forall x, y \in M : x \leq y \lor y \leq x$
  (Alle Elemente müssen in einer Relation zueinander stehen)

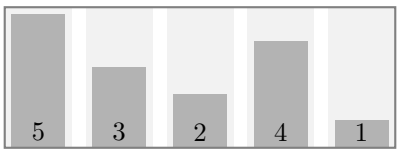
4.2 Insertion Sort

```java
class InsertionSort {
    void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            // 1 to n - 1
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                // Loops backwards through the array starting at i - 1
                // until it finds an element that is greater than the key or the beginning of the array
                arr[j + 1] = arr[j];
                // Shifts the element to the right
                j--;
            }
            arr[j + 1] = key;
            // Assigns the key to the correct position
        }
    }
}
```

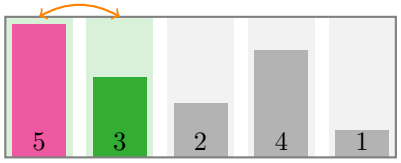
4.2 (a) Vorgehensweise

Die Eingabe wird von links nach rechts durchlaufen startend bei $i = 1$. Das Element i wird dann mit allen Element 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.

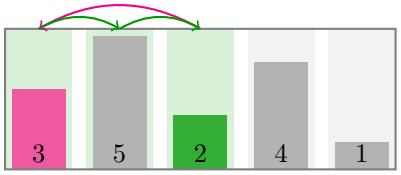
4.2 (b) Visuelle Darstellung



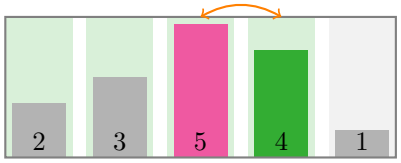
Anfangszustand



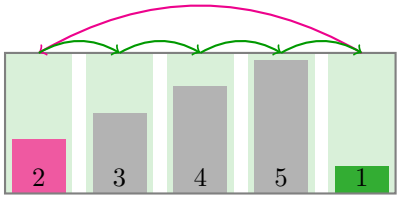
1. Iteration



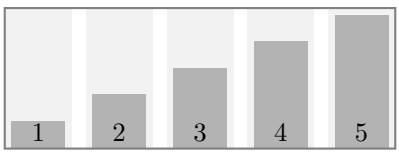
2. Iteration



3. Iteration



4. Iteration



Endzustand

Grün ist das momentan betrachte Element/Bereich. Magenta der Einfügepunkt des Elements.

4.2 (c) Komplexität

- **Worst-Case:**
  - Der Worst-Case ist ein array, der in reverse order sortiert ist.
  - Demnach muss jedes Element den kompletten array durchlaufen.
  - Dies ergibt eine Worst-Case Laufzeit von $\Theta(n^2)$
- **Best-Case:**
  - Der Best-Case ist ein array, 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 array, der in random order sortiert ist.
  - Demnach muss für jedes Element der array durchschnittlich bis zur Hälfte durchlaufen werden.
  - Nach der quadratischen Steigerung für große Zahlen ist die Hälfte aber irrelevant, weswegen $\Theta(n^2)$ ist.

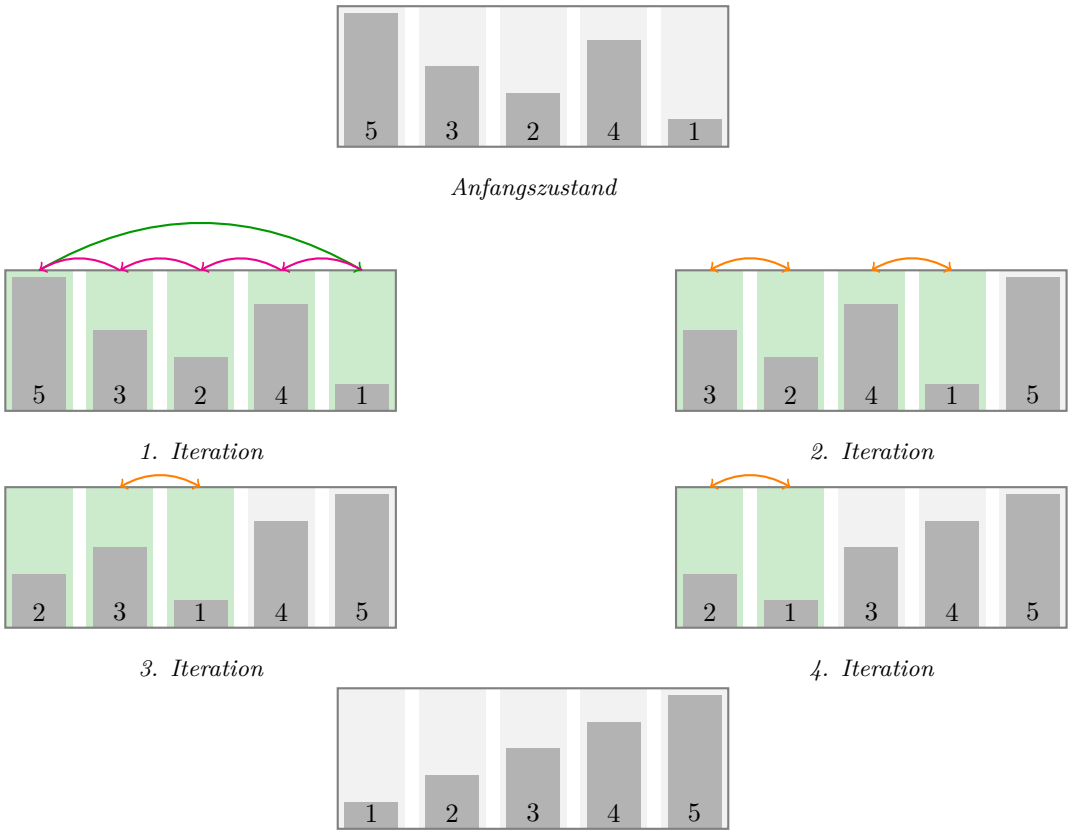
4.3 Bubble Sort

```java
class BubbleSort {

    void bubbleSort(int[] arr) {
        for(int i = arr.length - 1; i > 0; i--) {
            // Runs from arr.length - 1 to 0 (non exclusive)
            // (i = 0 would immediately terminate)
            boolean sorted = true;
            for(int j = 0; j < i; j++) {
                // Runs from 0 to i - 1
                if(arr[j] > arr[j + 1]) {
                    // If the current element is greater than the next
                    // Swap them
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    sorted = false;
                }
            }
            if(sorted)
                break;
        }
    }
}
```

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 Elemente 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 weitern 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 inefektiver 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 $\implies \Theta(n^2)$)
- **Average-Case:**
  - Die Eingabe ist zufällig sortiert.
  - Im Durchschnitt müssen die Elemente demnoch in den meisten Fällen getauscht werden.
  - Die Komplexität beträgt also $\Theta(n^2)$

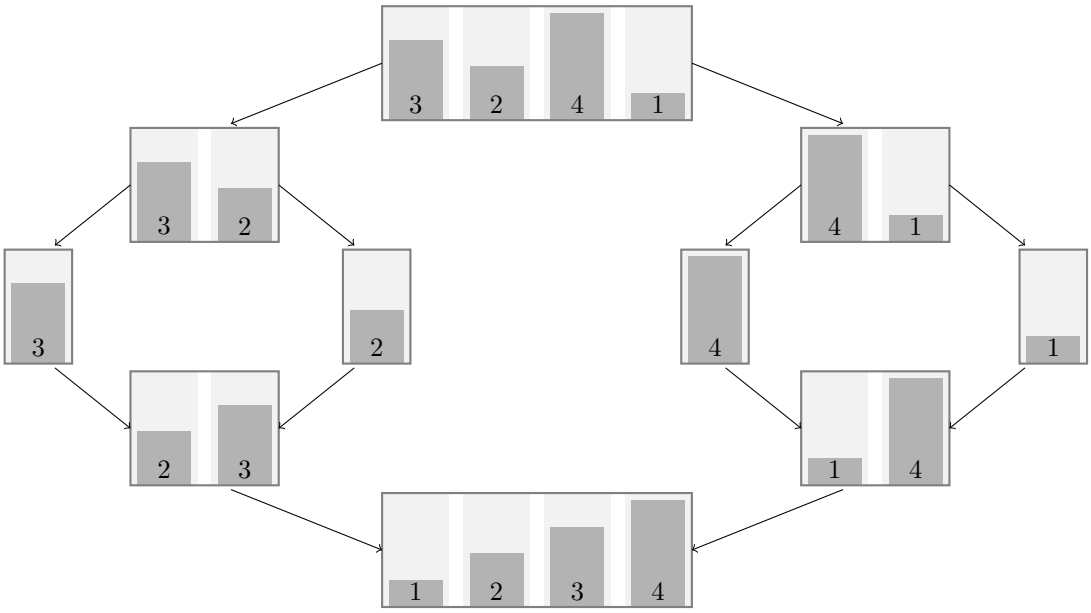
4.4 Merge Sort

```java
class MergeSort {
    void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            // left < right, otherwise the region has no elements
            int mid = (left + right) / 2; // Integer division -> round down
            // Split the region into two halves and do the recursive calls
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            // Merge the two (now sorted) halves
            merge(arr, left, mid, right);
        }
    }

    private void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        // Create a temporary array to store the merged elements

        int p = left;
        int q = mid + 1;
        for (int i = 0; i < right - left + 1; i++) {
            // Loops for each element in the region
            if (q > right || (p <= mid && arr[p] <= arr[q])) {
                // If p > mid the left half is finished, therefore the element needs to be in right half
                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
                temp[i] = arr[p];
                p++;
                // Adds the element at p to the temporary array and increases p
            }
            else {
                temp[i] = arr[q];
                q++;
                // Adds the element at q to the temporary array and increases q
            }
        }
        // Copy the merged elements from the temporary array back to the original array
        for (int i = 0; i < right - left + 1; i++)
            arr[left + i] = temp[i];
            // left + 0 is the start of the region
    }
}
```

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 enstandenen 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

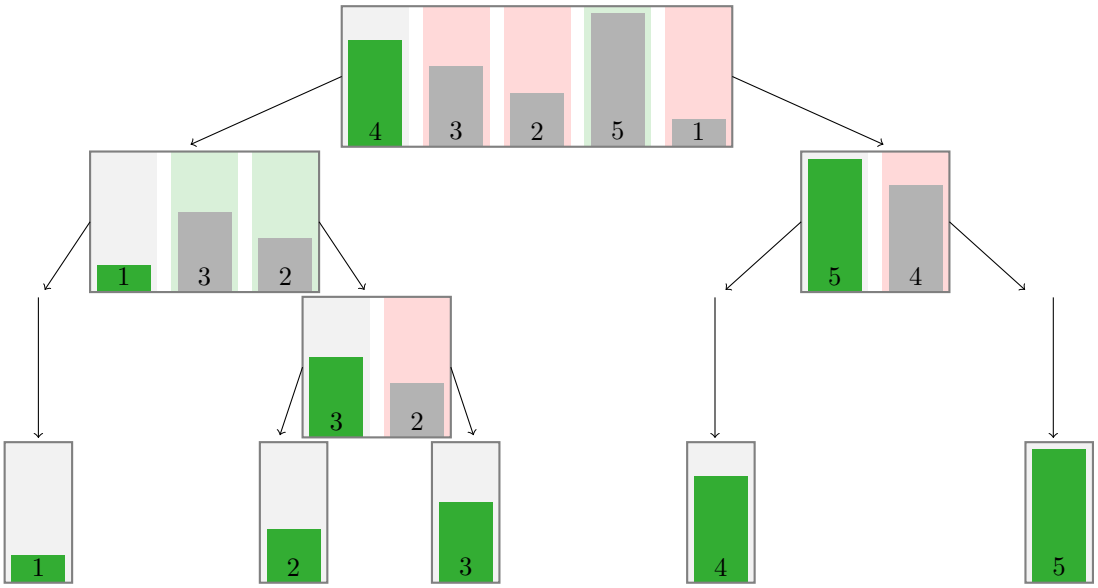
- **Worst-Case:**
  - Der Algorithmus funktioniert unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Worst-Case Eingabe.
  - Die Eingabe kann $\log n$ ($\log_2 n$)mal in zwei aufgeteilt werden kann. Zusätzlich benötigt der Algorithmus zum Kombinieren von den Teillisten n
  - Es ergibt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

```java
class Quicksort {
    void quickSort(int[] arr, int left, int right) {
        if (left < right) {
            // Region contains more than one element
            int part = partition(arr, left, right);
            quickSort(arr, left, part);
            quickSort(arr, part + 1, right);
        }
    }

    private int partition(int[] arr, int left, int right) {
        int pivot = arr[left];
        // Pivot is the first element in the region

        int p = left - 1;
        int q = right + 1;
        while (p < q) {
            do p++; while (arr[p] < pivot);
            do q--; while (arr[q] > pivot);
            // Increase / decrease p and q until the elements are bigger/smaller-equal pivot
            if (p < q) {
                /* p < q here means that theres a number bigger equal pivot on the left side
                and a number smaller equal than the pivot on the right side
                Therefore, we swap them to sort them into their halves*/
                int temp = arr[p];
                arr[p] = arr[q];
                arr[q] = temp;
                // Swap arr[p] and arr[q]
            }
        } /* This loop runs until p and q cross each other
        which means that */
        return q;
        // q is the index at which:
        // all indices greater than q contain elements bigger equal pivot
        // all indices smaller equal q contain elements smaller equal pivot
    }
}
```

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 unausgeglichenen 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 $\Theta(n \log n)$

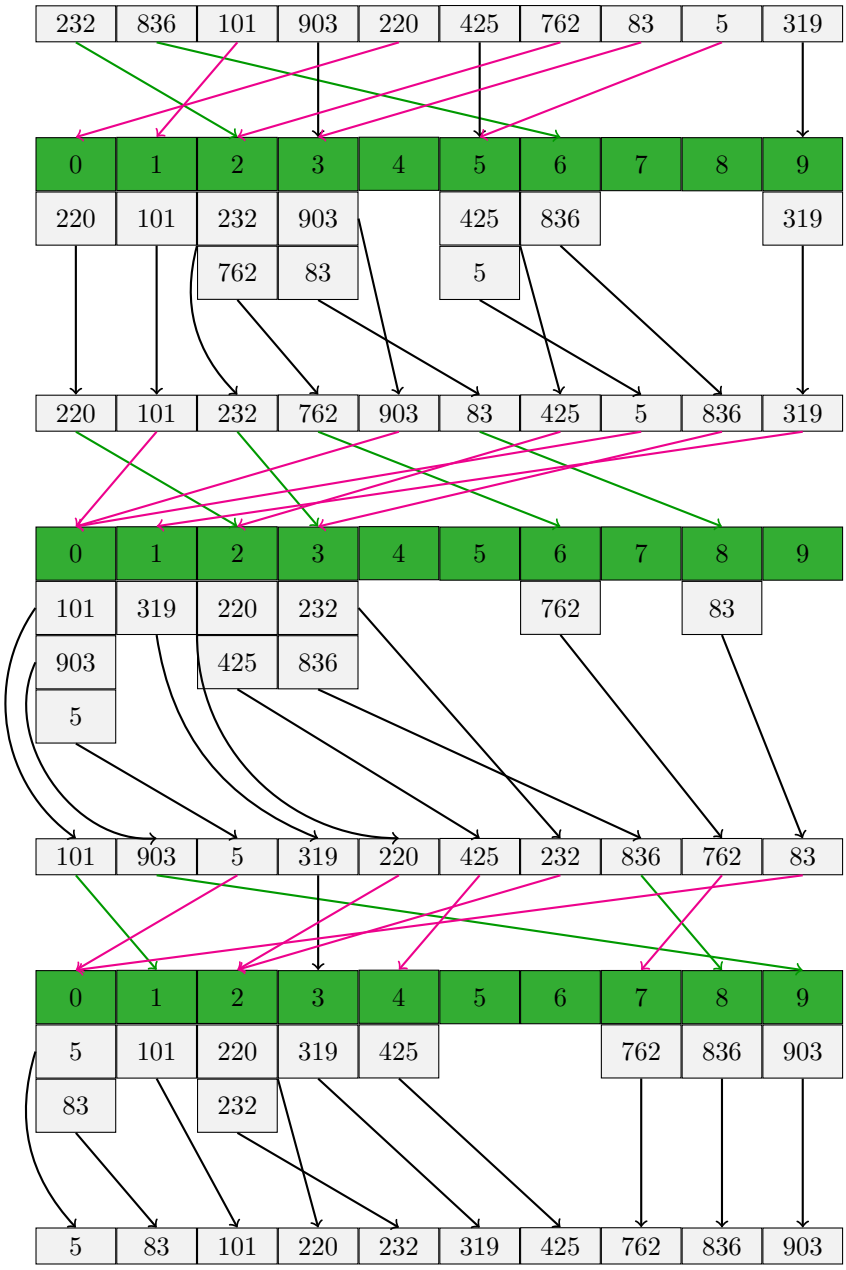
4.6 Radix Sort

```java
import java.util.ArrayList;

class RadixSort {
    int D = 10; // possible unique digits
    int d; // Max amount of digits
    ArrayList<Integer>[] buckets = new ArrayList[D];

    void radixSort(int[] arr) {
        d = amountDigits(arr);
        for (int i = 0; i < d; i++) {
            // for each digit in the array, 0 least significant
            for (int j = 0; j < arr.length; j++)
                putBucket(arr, i, j);
            // Sorts the numbers into their buckets
            int a = 0;
            for (int k = 0; k < D; k++) {
                for (int b = 0; b < buckets[k].size(); b++) {
                    arr[a] = buckets[k].get(b);
                    a++;
                }
                buckets[k].clear();
            }
            // Reads out the buckets in order
        }
    }

    private void putBucket(int[] arr, int i, int j) {
        int z = arr[j] / (int) Math.pow(D, i) % D;
        // Gets the ith digit of the number
        int b = buckets[z].size();
        // size is next free index
        buckets[z].add(b, arr[j]);
        // puts the number in the bucket z at position b
        // Depending on implementation might need to increase size manually
    }

    private int amountDigits(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max)
                max = arr[i];
        }
        // Get the biggest number
        return (int) (Math.log(max)/Math.log(D) + 1);
        // Get the amount of digits of the number
        // log(max)/log10(D) is equal to log_D(max)
    }
}
```

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

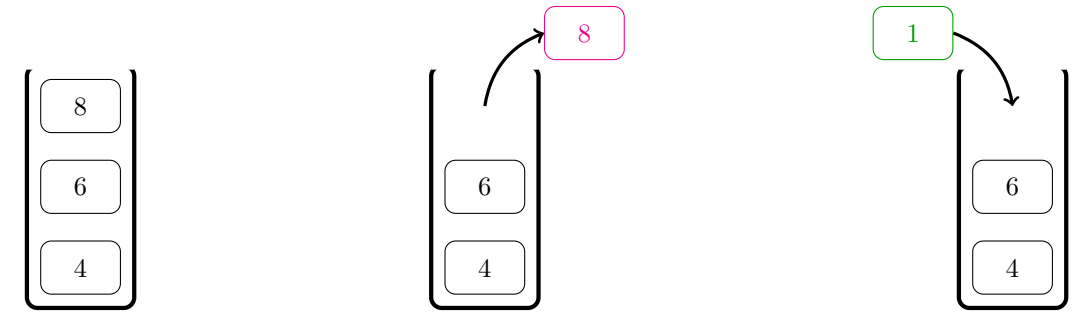
Stacks operieren unter dem ”First in - Last out” (FILO) Prinzip. Ähnlich zu einem Kartendeck, wo die unterste (Erste Karte) die ist, die als letztes gezogen wird.
Stacks werden normalerweise mit den folgenden Funktionen erstellt:

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

Eine mögliche Implementation auf Grundlage eines Arrays wäre:

```
class Stack {
    private int[] arr;
    private int top;
    Stack(int size) {
        arr = new int[size];
        top = -1;
        // Creates a new array with size
    }
    boolean isEmpty() {
        return top < 0;
        // Returns true if empty
    }
    int pop() { // O(1)
        return arr[top--];
        // Removes and returns the top element
    }
    void push(int k) { // O(1)
        arr[++top] = k;
        // Adds an element
    }
}
```

Push und Pop schmeißen Fehlermeldung wenn Stack leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wär es automatisch IndexOutOfBounds.
Oft werden Stacks auch mit variabler Größer 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.



Example Stack *Pop* *Push*

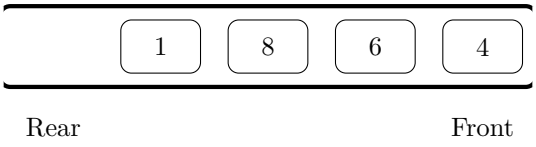
5.2 Queues

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:

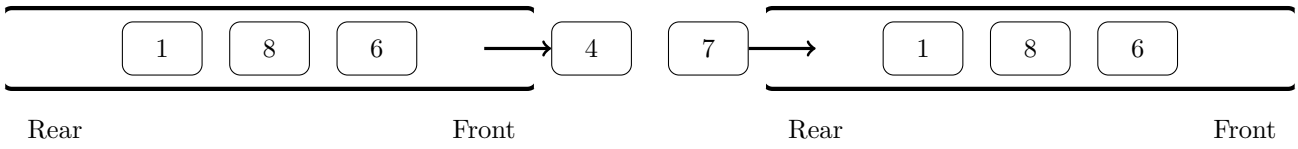
- `new(n)`: Erstellt einen neuen Queue.
- `isEmpty`: gibt an ob der Queue leer ist.
- `enqueue(k)`: Fügt `k` auf den Queue hinzu
- `dequeue`: gibt das erste Element des Queues zurück und entfernt es vom Queue.

Hier ist die Implementation für Queues wie folgt:

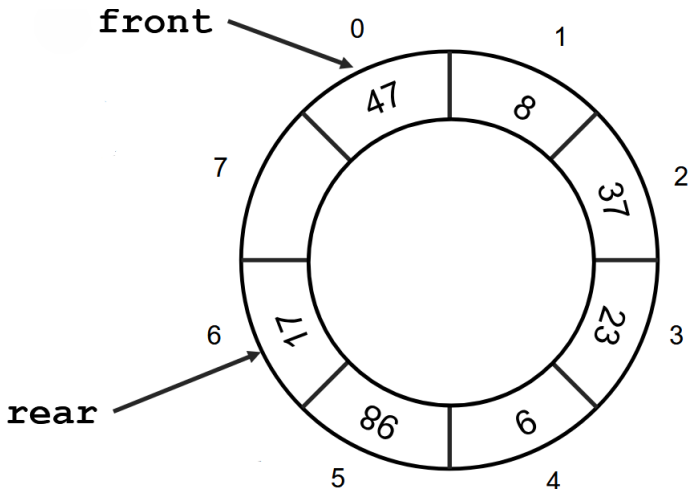
```
class Queue {
    private int[] arr;
    private int front;
    private int back;

    Queue(int size) {
        arr = new int[size];
        front = -1;
        back = -1;
    }

    boolean isEmpty() {
        return back == -1;
    }

    boolean isFull() {
        return (front + 1) % arr.length == back;
        // If front + 1 is equal to back, the queue is full
        // Modulo makes this usable for cyclic arrays
    }

    void enqueue(int k) { // O(1)
        if (isFull()) {
            throw new UException("Queue is full");
        } else{
            if (isEmpty())
                front = 0;
            back = (back + 1) % arr.length;
            // Modulo so that cyclic arrays work
            arr[back] = k;
        }
    }

    int dequeue() { // O(1)
        if (isEmpty()) {
            throw new UException("Queue is empty");
        } else {
            int temp = arr[front];
            front = (front + 1) % arr.length;
            // Modulo so that cyclic arrays work
            if (front == back) {
                front = -1;
                back = -1;
            }
            // If front and back are equal, the queue is empty -> reset
            return temp;
        }
    }
}
```



Example non-cyclic Queue



Dequeue *Enqueue*



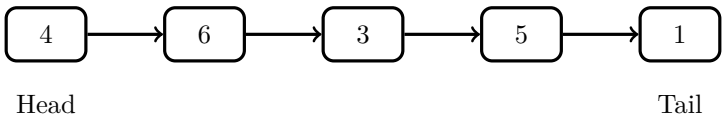
Cyclic Queue

5.3 Linked List

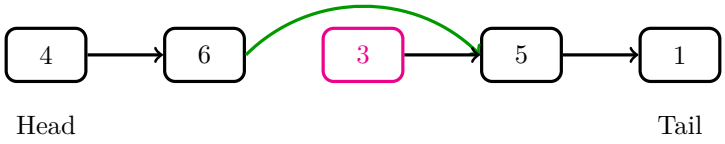
Eine einfache Linked List besteht aus mehreren Elementen, die jeweils immer einen Wert und eine Referenz auf das nächste Element in der Liste haben. Diese Struktur hat den Vorteil, dass sie keine festgelegte Größe hat, das Einfügen in $O(1)$ stattfindet, einfach zu implementieren ist und im Speicher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

```
class LinkedList {
    class LinkedElement {
        Integer key = null;
        LinkedElement next = null;

        LinkedElement(Integer key) {
            this.key = key;
        }
    }
    LinkedElement head = null; // First element in list
    LinkedElement tail = null; // Last element in list

    void insert(int k) { // O(1)
        LinkedElement elem = new LinkedElement(k);
        if (head == null) {
            head = elem;
            tail = elem;
        }
        else {
            tail.next = elem;
            tail = elem;
        }
    }

    void delete(int k) { // O(n)
        LinkedElement prev = null;
        LinkedElement curr = head;
        while (curr != null && curr.key != k) {
            prev = curr;
            curr = curr.next;
        }
        if (curr == null)
            throw new UException("Element not found");

        if (prev != null) {
            prev.next = curr.next;
            if (curr == tail)
                tail = prev;
        } else {
            head = curr.next;
        }
    }

    LinkedElement search(int k) { // O(n)
        LinkedElement curr = head;
        while (curr != null && curr.key != k)
            curr = curr.next;
        if (curr == null)
            throw new UException("Element not found");
        return curr;
    }
}
```

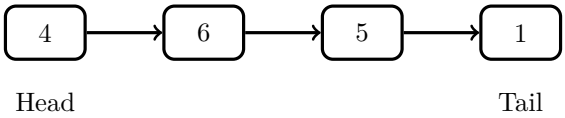
Diese Implementation benutzt einen Head und Tail, hat aber nur Referenz für das nächste Element in der Liste. Eine alternative Implementation wäre Tail wegzulassen und den Nodes eine previous-Referenz zu geben. Damit könnte man beim Einfügen das Element vorne an den Head anzuhängen und die neue Node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



Linked List

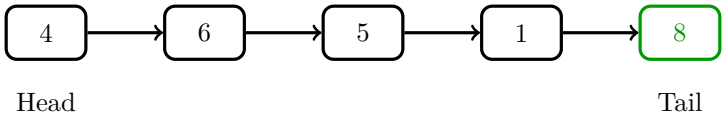


Delete Outcome



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.



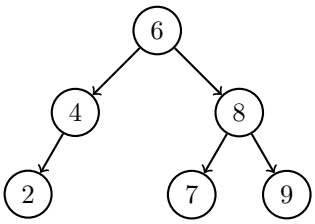
Insert of 8

8 wird an tail angehängt und wird dann zum tail

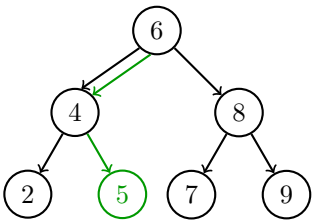
5.4 Binary Search Tree

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 balanziert, 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 unbalanziert, so dass der Worst-Case so aussieht, dass alle Nodes jeweils maximal ein Kind haben. Dies wäre effektiv gleich einer LinkedList.

```
class BSTree {
    class BSTNode {
        Integer key;
        BSTNode left;
        BSTNode right;
        BSTNode parent;
        BSTNode(Integer k) {
            key = k;
        }
    }
    BSTNode root;
    void insert(BSTNode z) { // Omega(1), O(h), Theta(h)
        BSTNode x = root; // Traversal starting from the root
        BSTNode px = null; // Parent of x, initially null
        while(x != null) {
            px = x;
            if (z.key < x.key)
                x = x.left;
            else
                x = x.right;
        }// Traversing the tree until finding the insertion point
        z.parent = px; // Sets the parent of the node to be inserted
        if (px == null) // px only null if the tree is empty-> loop never runs -> z is root
            root = z;
        else if (z.key < px.key) // Key smaller -> left child
            px.left = z;
        else // Key bigger -> right child
            px.right = z;
        // May add the same node twice as it doesn't check for duplicates
    }
```

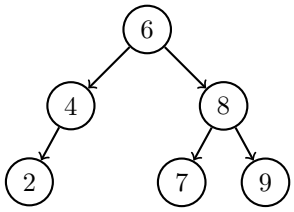


Before insert

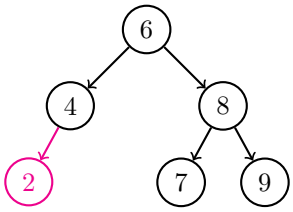


Insert 5

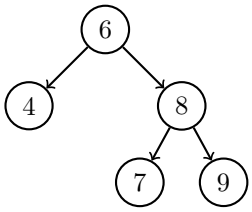
```
    void delete(BSTNode z) { // Omega(1), O(h), Theta(log n)
        if (z.left == null) // If z has no left, transplants the right to z's position
            transplant(z, z.right);
        else if (z.right == null) // If z has no right, transplants the left to z's position
            transplant(z, z.left);
        else { // If z has both left and right children
            BSTNode y = z.right;
            while (y.left != null)
                // Finds the next biggest element of z = smallest in right subtree of z
                y = y.left;
            if (y.parent != z) { // If the next biggest element y is not child of z
                transplant(y, y.right); // Transplants the right child of y to y's position
                y.right = z.right; // The right child of y becomes the right child of z
                y.right.parent = y; // The parent of the right child of y becomes y
            }
            transplant(z, y); // Transplants y to z's position
            y.left = z.left; // The left child of y becomes the left child of z
            y.left.parent = y; // The parent of the left child of y becomes y
        }
    }
    void transplant(BSTNode u, BSTNode v) { // O(1)
        // Transplants v to the parent of u
        if (u.parent == null) // If u is the root, v becomes the new root
            root = v;
        else if (u == u.parent.left) // If u is a left child, v becomes a left child
            u.parent.left = v;
        else // If u is a right child, v becomes a right child
            u.parent.right = v;
        if (v != null) // If v is not null, v becomes a child of u's parent
            v.parent = u.parent;
    }
```



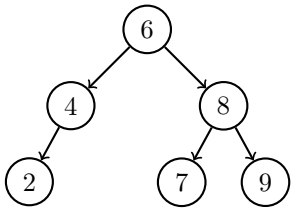
Leaf Deletion



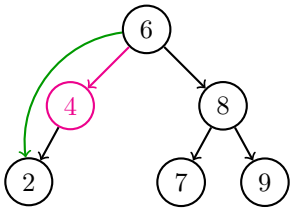
Delete 2



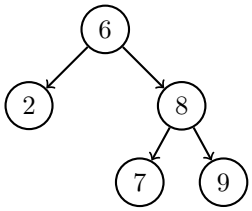
Result



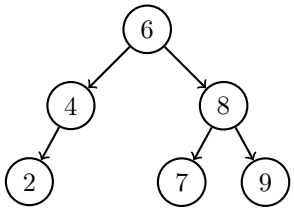
Half-Leaf Deletion



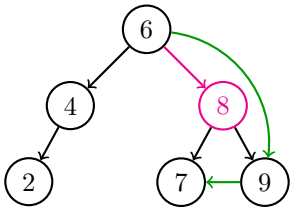
Delete 4



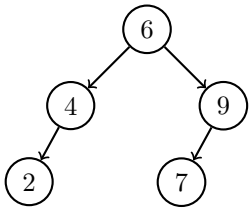
Result



Complete Node Deletion



Delete 8

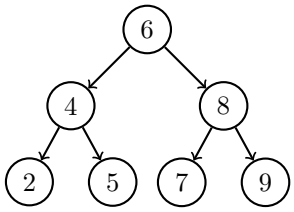


Result

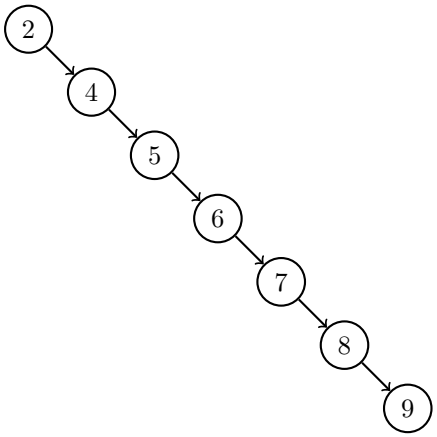
Leaves werden gelöscht, Half-Leaves durch Kind ersetzt, Complete Node durch Nachfolger (nächstgrößtes Element, kleinstes Element im rechten Teilbaum der Node) ersetzt.

```
    BSTNode iterativeSearch(int k) { // O(h), Omega(1), Theta(log n)
        BSTNode curr = root;
        while (curr != null && curr.key != k) {
            if (k < curr.key)
                curr = curr.left;
            else
                curr = curr.right;
        }
        return curr; // Returns null if element not found
    }
    BSTNode recursiveSearch(int k, BSTNode curr) {// O(h), Omega(1), Theta(log n)
        if (curr == null)
            return null;
        if (k < curr.key)
            return recursiveSearch(k, curr.left);
        else if (k > curr.key)
            return recursiveSearch(k, curr.right);
        return curr; // Returns null if element not found
    }
```

```
    void traversal(BSTNode curr) { // O(n)
        if (curr != null)
            return;
        // Any actions that should be done in a specific order can be done
        // Here for preorder traversal
        traversal(curr.left);
        // Here for inorder traversal
        traversal(curr.right);
        // Here for postorder traversal
        // Left and right can also be exchanged to traverse in reverse order
    }
}
```



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



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

6 Fortgeschrittene Datenstrukturen

6.1 Red-Black Tree

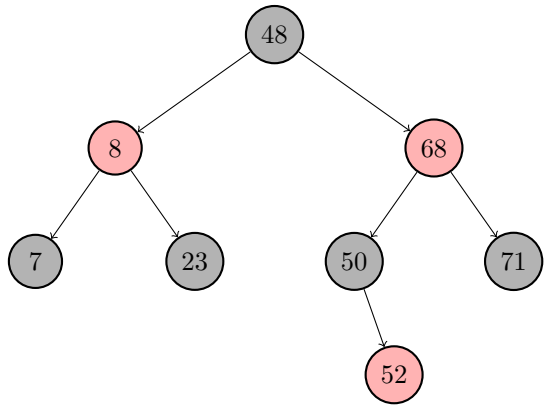
Ein Red-Black Tree ist eine Art Binary-Search Tree. Zusätzlich zu diesem besitzen die Nodes in einem RB Tree noch das Attribut `color`. Die Nodes werden also entweder als `red` oder `black` definiert. Dies dient zur Einhaltung der Red-Black-Regeln, durch die die Effizienz der Datenstruktur im Vergleich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

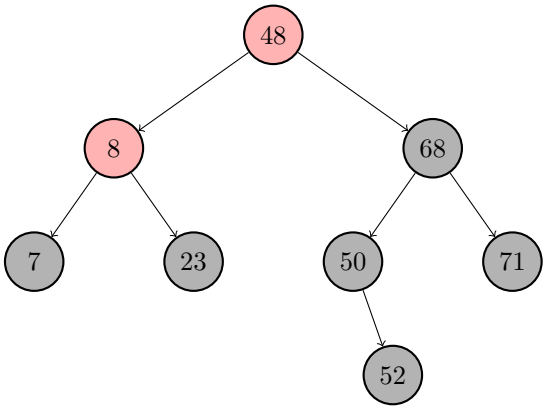
$\Longrightarrow$ Hat ein Knoten nur ein Kind, so muss dieses Kind Rot sein, ansonsten ist die Anzahl an schwarzen Knoten auf dem Pfad unterschiedlich zu den anderen Pfaden.

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.

```
class RBTree {
    class RBNode {
        Integer key;
        RBNode left;
        RBNode right;
        RBNode parent;
        Color color;
        RBNode(Integer k) {
            key = k;
        }
    }
    RBNode sent;
    RBNode root = null;
    RBTree() {
        sent = new RBNode(null);
        sent.color = Color.BLACK;
        sent.left = sent;
        sent.right = sent;
        // Sentinel always points to itself ->
        // node.parent.parent and its children will never result in null references
    }
    // Traversal and search are the same as BSTree
```



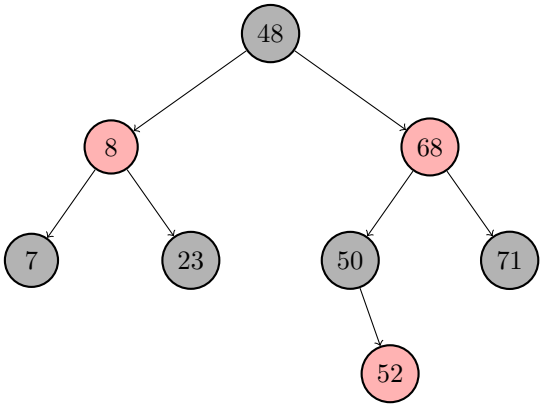
Richtig Konstruierter RBT



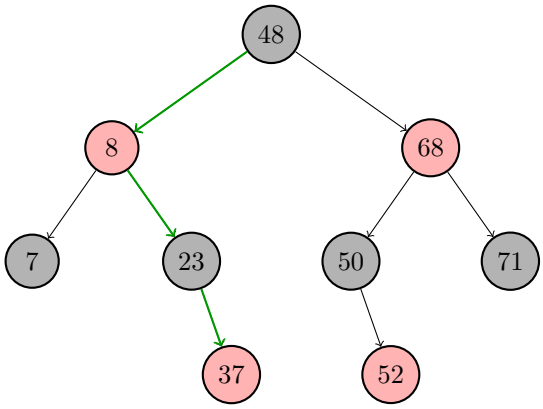
Falsch Konstruierter RBT

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

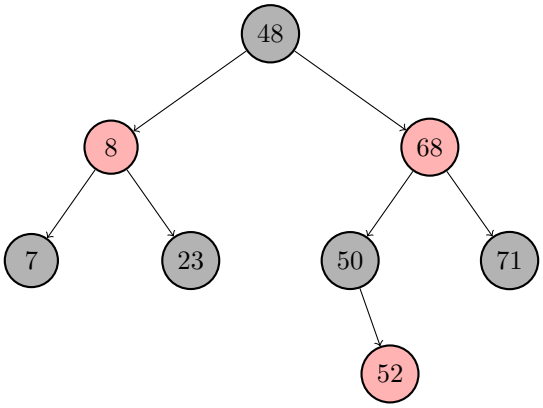
Einfügen funktioniert grundlegend gleich zu BST, allerdings wird am Ende die Farbe des neuen Knotens auf rot gesetzt und anschließend die Regeln des RBTs (falls verletzt) wieder hergestellt.



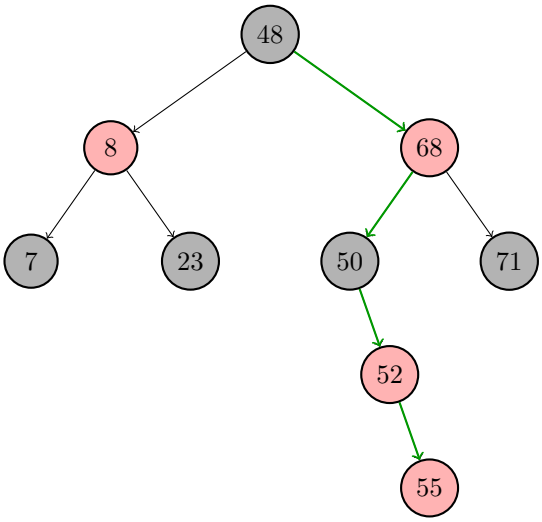
Before



Insert 37 (No colorfix needed)

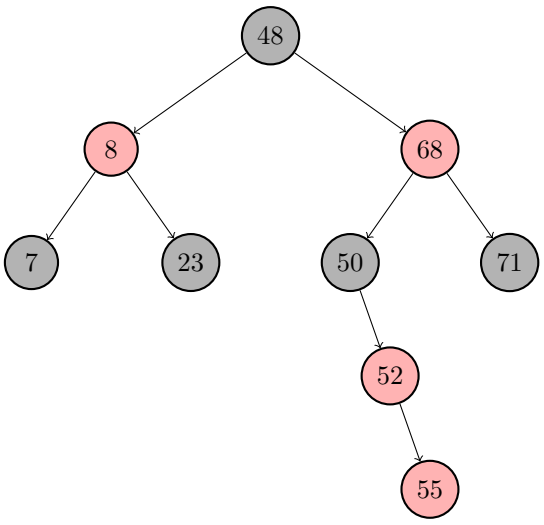


Before

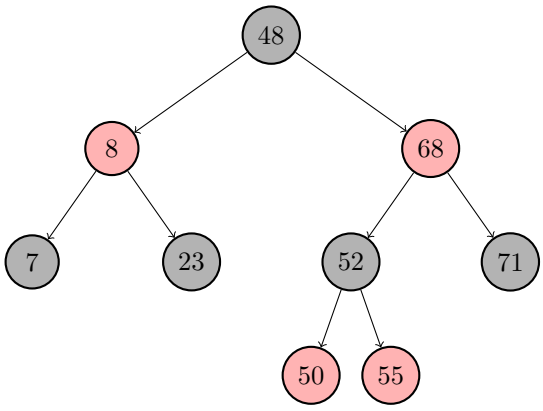


Insert 55 (Colorfix needed)

```java
void fixColorsAfterInsertion(RBNode z) { // Omega(1), O(log n), Theta(log n)
    while (z.parent.color == Color.RED) { // While z's parent is red
        if (z.parent == z.parent.parent.left) { // If z's parent is a left child
            RBNode y = z.parent.parent.right; // Gets sibling of z's parent
            if (y != null && y.color == Color.RED) { // CASE 1: z's parent is a left child and sibling is red
                z.parent.color = Color.BLACK; // Set z's parent to black
                y.color = Color.BLACK; // Set z's uncle to black
                z.parent.parent.color = Color.RED; // Set z's grandparent to red
                z = z.parent.parent; // Set z to z's grandparent
            } else { // CASE 2: z's parent is a left child and sibling is black
                if (z == z.parent.right) { // If z is a right child
                    z = z.parent; // Set z to z's parent
                    rotateLeft(z); // Rotate new z to left
                }
                z.parent.color = Color.BLACK; // Set z's parent to black
                z.parent.parent.color = Color.RED; // Set z's grandparent to red
                rotateRight(z.parent.parent); // Rotate z's grandparent to right
            }
        } else { // If z's parent is a right child
            // Same as above but with right and left exchanged
            RBNode y = z.parent.parent.left;
            if (y != null && y.color == Color.RED) { // CASE 3: z's parent is a right child and sibling is red
                z.parent.color = Color.BLACK;
                y.color = Color.BLACK;
                z.parent.parent.color = Color.RED;
                z = z.parent.parent;
            } else { // CASE 4: z's parent is a right child and sibling is black
                if (z == z.parent.left) {
                    z = z.parent;
                    rotateRight(z);
                }
                z.parent.color = Color.BLACK;
                z.parent.parent.color = Color.RED;
                rotateLeft(z.parent.parent);
            }
        }
    }
    root.color = Color.BLACK; // Set root to black, as it always should be
}
```



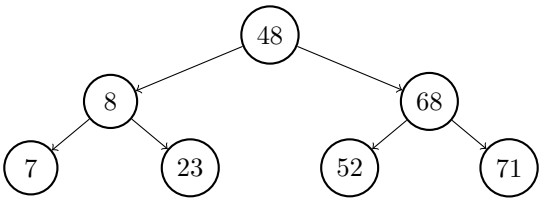
Fix needed at 55



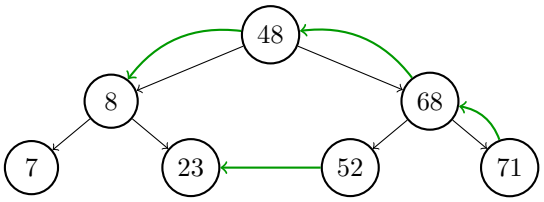
Fixed (Case 4)

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

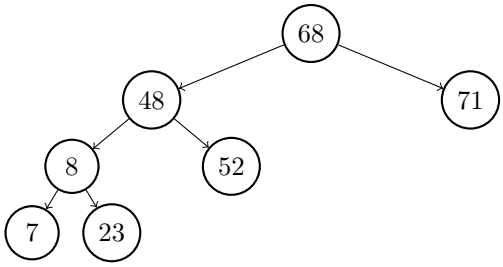
```
    void rotateLeft(RBNode x) { // O(1)
        RBNode y = x.right;
        x.right = y.left; // Set x's right child to y's left child
        if (y.left != null) // If y has a left child
            y.left.parent = x; // Set y's left child's parent to x
        y.parent = x.parent; // Set y's parent to x's parent
        if (x.parent == sent) // If x is the root, set y to be the root
            root = y;
        else if (x == x.parent.left) // If x is a left child, set x's parent's left child to y
            x.parent.left = y;
        else // If x is a right child, set x's parent's right child to y
            x.parent.right = y;
        y.left = x; // Set y's left child to x
        x.parent = y; // Set x's parent to y
    }
    void rotateRight(RBNode x) { // O(1)
        // Same as rotateLeft but with right and left exchanged
        RBNode y = x.left;
        x.left = y.right;
        if (y.right != null)
            y.right.parent = x;
        y.parent = x.parent;
        if (x.parent == sent)
            root = y;
        else if (x == x.parent.right)
            x.parent.right = y;
        else
            x.parent.left = y;
        y.right = x;
        x.parent = y;
    }
```



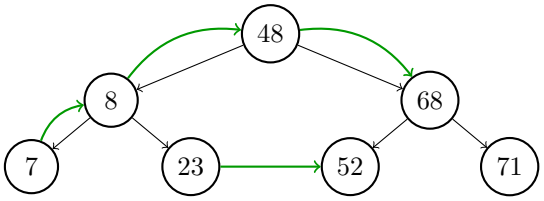
Before



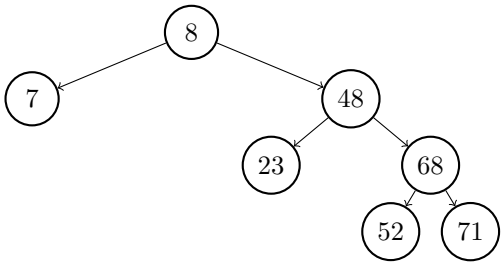
rotateLeft(root)



rotateLeft(root) result

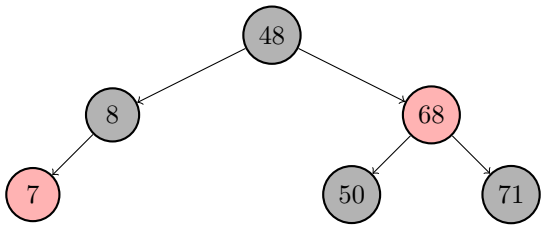


rotateRight(root)

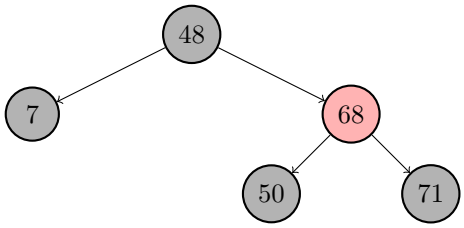


rotateRight(root) result

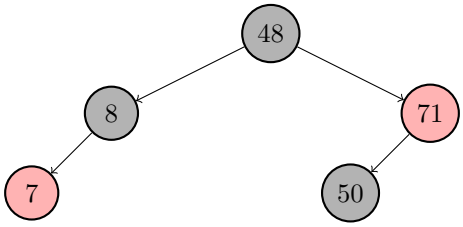
```
    void delete(RBNode z) { // Omega(1), O(log n), Theta(log n)
        RBNode a = z.parent; // a represent node with black depth imbalance
        int dbh = 0; // delta black height, -1 for right, 1 for left leaning
        if (z.left == null && z.right == null) { // CASE 1: z is a leaf
            if (z.color == Color.BLACK && z != root) { // If z is black
                if (z == z.parent.left) // If z is a left child
                    dbh = -1; // Set delta black height to -1
                else // If z is a right child
                    dbh = 1; // Set delta black height to 1
            }
            transplant(z, null); // Transplant null to zs parent
        } else if (z.left == null) { // CASE 2: z only has a right child
            RBNode y = z.right;
            transplant(z, z.right);
            y.color = z.color;
        } else if (z.right == null) { // CASE 3: z only has a left child
            RBNode y = z.left;
            transplant(z, z.left);
            y.color = z.color;
        } else { // CASE 4: z has two children
            RBNode y = z.right;
            a = y;
            boolean wentLeft = false;
            while (y.left != null) { // find next biggest Node
                a = y;
                y = y.left;
                wentLeft = true;
            }
            if (y.parent != z) { // If next biggest element is not child of z
                transplant(y, y.right);
                y.right = z.right;
                y.right.parent = y;
            }
            transplant(z, y);
            y.left = z.left;
            y.left.parent = y;
            if (y.color == Color.BLACK) {
                if (wentLeft) // Tree imbalanced depending on y location
                    dbh = -1;
                else
                    dbh = 1;
            }
            y.color = z.color;
        }
        if (dbh != 0) // If black height imbalance
            fixColorsAfterDeletion(a, dbh);
    }
```



Before



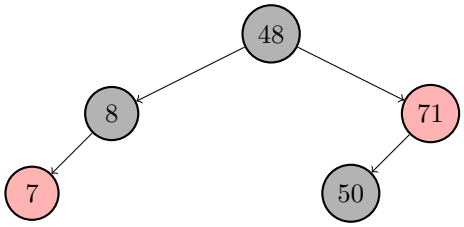
Delete 8 (No colorfix needed, Case 3)



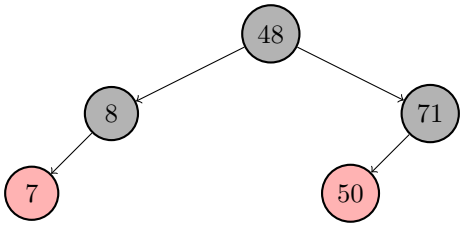
Delete 68 (Colorfix needed, Case 4, dbh = 1, skewed left)

```
    void fixColorsAfterDeletion(RBNode a, int dbh) { // Omega(1), O(log n), Theta(log n)
        if (dbh == -1) { // Extra black node on the right
            RBNode x = a.left;
            RBNode b = a.right;
            RBNode c = b.left; // Left child of right child of a
            RBNode d = b.right; // Right child of right child of a
            if (x != null && x.color == Color.RED) {
                // Easy case: x is red
                x.color = Color.BLACK;
            } else if (a.color == Color.BLACK
                    && b.color == Color.RED) {
                // Case 1: a black, b red
                rotateLeft(a);
                a.color = Color.RED;
                b.color = Color.BLACK;
                fixColorsAfterDeletion(a, dbh);
            } else if (a.color == Color.RED
                    && b.color == Color.BLACK
                    && (c == null || c.color == Color.BLACK)
                    && (d == null || d.color == Color.BLACK)) {
                // Case 2a: a red, b black, c and d black
                a.color = Color.BLACK;
                b.color = Color.RED;
            } else if (a.color == Color.BLACK
                    && b.color == Color.BLACK
                    && (c == null || c.color == Color.BLACK)
                    && (d == null || d.color == Color.BLACK)) {
                // Case 2b: a black, b black, c and d black
                b.color = Color.RED;
                if (a == a.parent.left)
                    dbh = 1;
                else if (a == a.parent.right)
                    dbh = -1;
                else
                    dbh = 0;
                fixColorsAfterDeletion(a.parent, dbh);
            } else if (b.color == Color.BLACK
                    && c != null && c.color == Color.RED
                    && (d == null || d.color == Color.BLACK)) {
                // Case 3: a either, b black, c red, d black
                rotateRight(b);
                c.color = Color.BLACK;
                fixColorsAfterDeletion(a, dbh);
            } else if (b.color == Color.BLACK
                    && d!= null && d.color == Color.RED) {
                // Case 4: a either, b black, c either, d red
                rotateLeft(a);
                b.color = a.color;
                a.color = Color.BLACK;
                d.color = Color.BLACK;
                }
```

```
        } else { // Extra black node on the left
            // Same as above but with right and left exchanged
            RBNode x = a.right;
            RBNode b = a.left;
            RBNode c = b.right; // Right child of left child of a
            RBNode d = b.left; // Left child of left child of a
            if (x != null && x.color == Color.RED) {
                // Easy case: x is red
                x.color = Color.BLACK;
            } else if (a.color == Color.BLACK
                    && b.color == Color.RED) {
                // Case 1: a black, b red
                rotateRight(a);
                a.color = Color.RED;
                b.color = Color.BLACK;
                fixColorsAfterDeletion(a, dbh);
            } else if (a.color == Color.RED
                    && b.color == Color.BLACK
                    && (c == null || c.color == Color.BLACK)
                    && (d == null || d.color == Color.BLACK)) {
                // Case 2a: a red, b black, c and d black
                a.color = Color.BLACK;
                b.color = Color.RED;
            } else if (a.color == Color.BLACK
                    && b.color == Color.BLACK
                    && (c == null || c.color == Color.BLACK)
                    && (d == null || d.color == Color.BLACK)) {
                // Case 2b: a black, b black, c and d black
                b.color = Color.RED;
                if (a == a.parent.right)
                    dbh = 1;
                else if (a == a.parent.left)
                    dbh = -1;
                else
                    dbh = 0;
                fixColorsAfterDeletion(a.parent, dbh);
            } else if (b.color == Color.BLACK
                    && c != null && c.color == Color.RED
                    && (d == null || d.color == Color.BLACK)) {
                // Case 3: a either, b black, c red, d black
                rotateLeft(b);
                c.color = Color.BLACK;
                fixColorsAfterDeletion(a, dbh);
            } else if (b.color == Color.BLACK
                    && d!= null && d.color == Color.RED) {
                // Case 4: a either, b black, c either, d red
                rotateRight(a);
                b.color = a.color;
                a.color = Color.BLACK;
                d.color = Color.BLACK;
            }
        }
        // All cases except 2b mean end of the method in this or the next instance. 2b can go on tho.
    }
```



Before fix



Fixed (Case 2a)

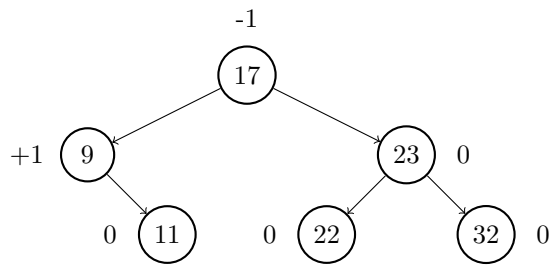
```
    void transplant(RBNode u, RBNode v) { // O(1)
        // Transplants v to the position of u
        if (u.parent == sent) // If u is the root, v becomes the new root
            root = v;
        else if (u == u.parent.left) // If u is a left child, v becomes a left child
            u.parent.left = v;
        else // If u is a right child, v becomes a right child
            u.parent.right = v;
        if (v != null) // If v is not null, v becomes a child of u's parent
            v.parent = u.parent;
    }
}
```

6.2 AVL Trees

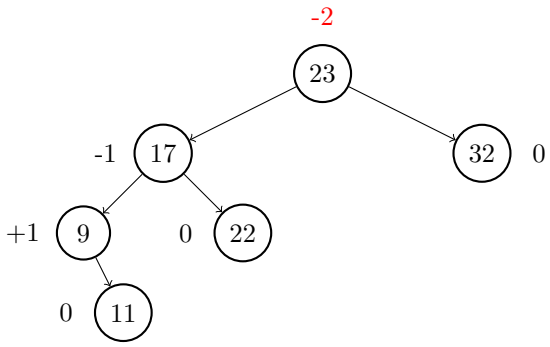
Ein **A**delson-**V**elsky **L**andis Tree ist ein BST, der sich selbst balanziert um eine Höhe von $h = \log n$ zu garantieren. Er wird so definiert, **dass die Höhendifferenz von zwei Teilbäumen unter einem Knoten jeweils maximal 1 ist**.
Im Vergleich zu RBT ist der AVLT in der Höhe strikter. So ist die maximale Höhe im AVL $h = 1.44 \cdot \log n$, während er im RBT nur $2 \cdot \log n$. So haben AVLTs zwar die bessere Effizienz in Suchvorgängen, jedoch benötigen sie beim Insert oder Delete meist mehr Rotationen. Demnach bieten sich AVLTs 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.

```
class AVLTree {
    class AVLNode {
        Integer key;
        int height;
        AVLNode left;
        AVLNode right;
        AVLNode(Integer k) {
            key = k;
            height = 1;
        }
    }
    AVLNode root;
    // Search and traversal like in BST
    int height(AVLNode n) {
        return (n == null) ? -1 : n.height;
    }
    void updateHeight(AVLNode n) {
        n.height = 1 + Math.max(height(n.left), height(n.right));
    }
    int getBalance(AVLNode n) {
        return (n == null) ? 0 : height(n.right) - height(n.left);
    }
```

Visuell gleich zu einem Best-Case ausgewogenen BST.



Balanced Tree



Unbalanced Tree

```
    // Rotations work a bit different as in other Trees as we don't have parents.
    // Does not update the tree itself -> just returns the new root of the rotated subtree
    // Does not need to check for null reference as right.left/left.right respectively
    // are always well defined when called in insert and delete
    AVLNode rotateLeft(AVLNode x) { // O(1)
        AVLNode y = x.right;
        AVLNode z = y.left;
        y.left = x;
        x.right = z;
        updateHeight(x);
        updateHeight(y);
        return y;
    }
    AVLNode rotateRight(AVLNode x) { // O(1)
        AVLNode y = x.left;
        AVLNode z = y.right;
        y.right = x;
        x.left = z;
        updateHeight(x);
        updateHeight(y);
        return y;
    }
```

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

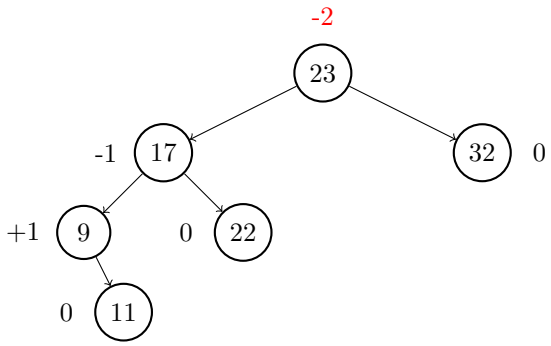
```
    AVLNode insert(AVLNode partRoot, int key) { // O(h) = O(log(n))
        if (partRoot == null) // Found insertion point
            return new AVLNode(key);
        else if (key < partRoot.key) // If node smaller than partRoot -> go left
            partRoot.left = insert(partRoot.left, key);
        else if (key > partRoot.key) // If node bigger than partRoot -> go right
            partRoot.right = insert(partRoot.right, key);
        else // If node == partRoot -> throw exception
            throw new UException("Duplicate node");
        return fixBalance(partRoot);
        // Rebalance every node above the inserted node.
    }
```

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

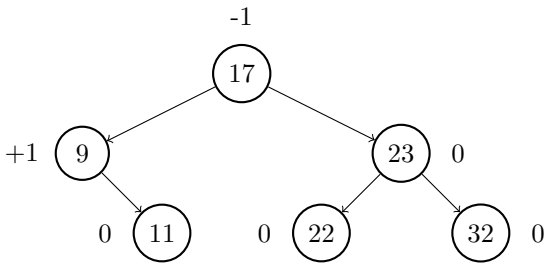
```
AVLNode delete(AVLNode node, int key) { // O(h) = O(log(n))
    if (node == null) {// Node not found
        return node;
    } else if (key < node.key) {
        node.left = delete(node.left, key);
    } else if (key > node.key) {
        node.right = delete(node.right, key);
    } else { // Node found -> Commence deletion
        if (node.left == null || node.right == null) // If half leaf or leaf
            node = (node.left == null) ? node.right : node.left;
        else { // If complete node
            AVLNode next = node.right;
            while (next.left != null) {
                next = next.left;
            }
            node.key = next.key;
            node.right = delete(node.right, next.key);
        }
    }
    return fixBalance(node); // Rebalance every node above the deleted node
}
```

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

```
    AVLNode fixBalance(AVLNode z) { // O(1)
        updateHeight(z);
        int balance = getBalance(z);
        if (balance > 1) { // If right heavy
            if (height(z.right.right) > height(z.right.left)) {
                z = rotateLeft(z);
            } else {
                z.right = rotateRight(z.right);
                z = rotateLeft(z);
            }
        } else if (balance < -1) { // If left heavy
            if (height(z.left.left) > height(z.left.right)) {
                z = rotateRight(z);
            } else {
                z.left = rotateLeft(z.left);
                z = rotateRight(z);
            }
        }
        return z;
    }
}
```



Unbalanced Tree after insert of 11



Balanced Tree after fixup

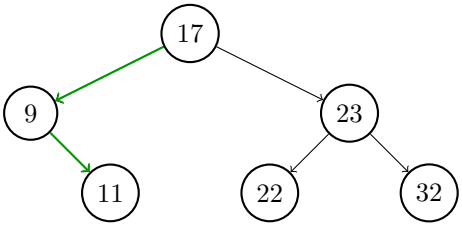
6.3 Splay Trees

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 AVLT 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.

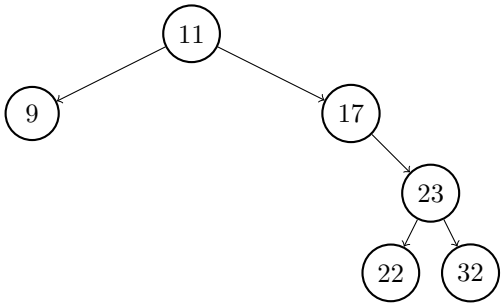
```
class SplayTree extends BSTree {
    //Same Nodes as BSTree
    void splay(BSTNode z) { // O(h)
        while(z != root) {
            if (z.parent.parent == null) // If father is root
                zig(z);
            else {
                if (z == z.parent.parent.left.left || z == z.parent.parent.right.right)
                    zigZig(z);
                else
                    zigZag(z);
            }
        }
    }
    void zig(BSTNode z) { // O(1)
        if (z == z.parent.left)
            rotateRight(z.parent);
        else
            rotateLeft(z.parent);
    }
    void zigZig(BSTNode z) { // O(1)
        if (z == z.parent.left) {
            rotateRight(z.parent.parent);
            rotateRight(z.parent);
        } else {
            rotateLeft(z.parent.parent);
            rotateLeft(z.parent);
        }
    }
    void zigZag(BSTNode z) { // O(1)
        if (z == z.parent.left) {
            rotateRight(z.parent);
            rotateLeft(z.parent.parent);
        } else {
            rotateLeft(z.parent);
            rotateRight(z.parent.parent);
        }
    }
    //Rotate works the same as in RBTree
```

Sieht visuell praktisch wie ein normaler BST aus.

```
    BSTNode search(int key) { // O(h), like BSTree, with additional splay
        BSTNode x = root;
        while(x != null && x.key != key) {
            if (key < x.key)
                x = x.left;
            else
                x = x.right;
        }
        if (x == null)
            return null;
        splay(x);
        return root; // After splay the root is the searched node
    }
```

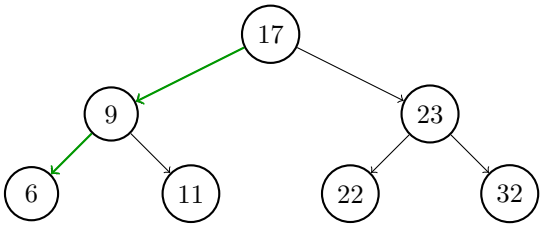


Search for 11

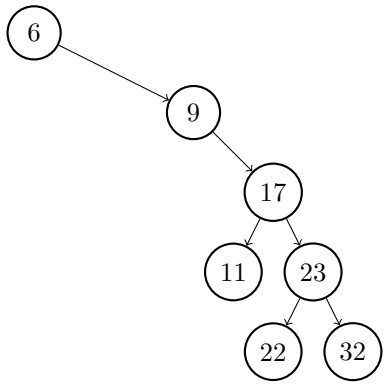


Splay 11 (zigzag)

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

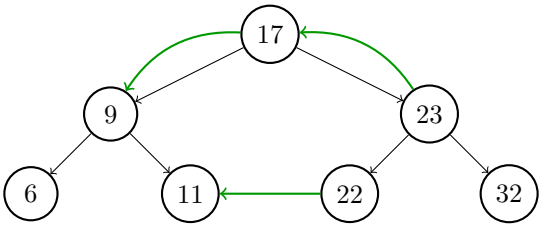


Insert 6

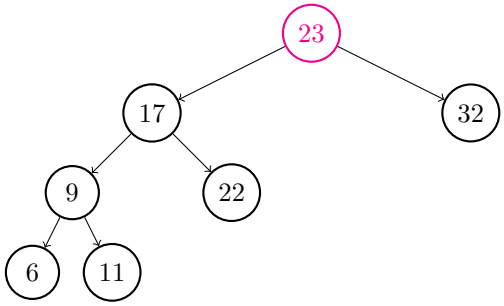


Splay 6 (zigzig)

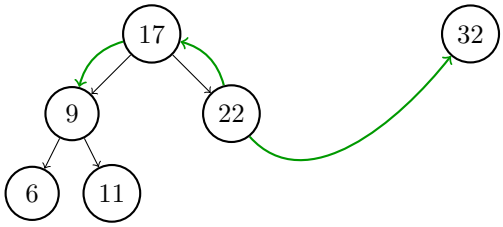
```
    void delete (BSTNode z) { // O(h)
        splay(z);
        BSTNode r = root.right; // Save right child
        BSTNode biggestL = root.left; // Save left child
        root.right.parent = null; // Remove parent reference of right child of root
        root.left.parent = null; // Remove parent reference of left child of root
        root = biggestL; // Set root to left child
        while (biggestL.right != null) // Get biggest node in left subtree
            biggestL = biggestL.right;
        splay(biggestL); // Splay biggest node -> becomes root
        root.right = r; // Put right subtree back in place
        r.parent = root; // Change parent of right subtree root
    }
}
```



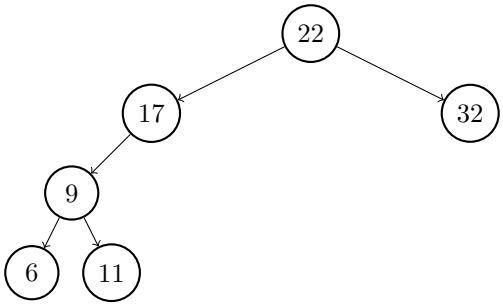
Before delete 23



Splay 23 (zig)



delete 23



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

6.4 Binary Heap Trees

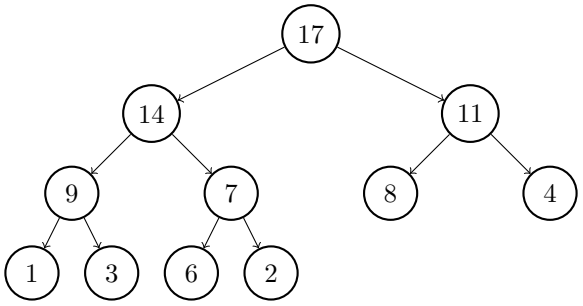
Im Allgemeinen ist ein Binary Heap wie folgt definiert:

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

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 organiesiert, 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**: $parent(i) = \lceil i/2 \rceil - 1$
- **Left Child**: $left(i) = 2 \cdot (i + 1) - 1$
- **Right Child**: $right(i) = 2 \cdot (i + 1)$

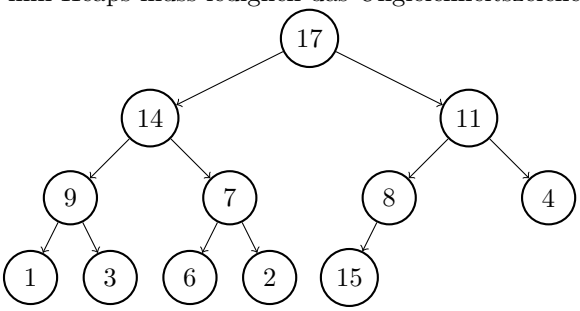
```java
class BinaryMaxHeap {
    Integer[] heap;
    int size; // Size used to insert new elements at the first empty index
    BinaryMaxHeap(int n) { // Creates new heap with size n
        heap = new Integer[n];
        size = 0;
    }
    BinaryMaxHeap(Integer[] arr) { // Creates heap from array
        arrayToHeap(arr);
    }
    void arrayToHeap(Integer[] arr) { // Used to transfer array to heap
        heap = arr.clone();
        size = 0;
        for (Integer i: arr) {
            if (i == null) break;
            size++;
        }
    }
    int parent(int i) {
        return (int) Math.ceil((double) i / 2) - 1;
    }
    int left(int i) {
        return 2 * (i + 1) - 1;
    }
    int right(int i) {
        return 2 * (i + 1);
    }
    boolean isEmpty() {
        return size == 0;
    }
```



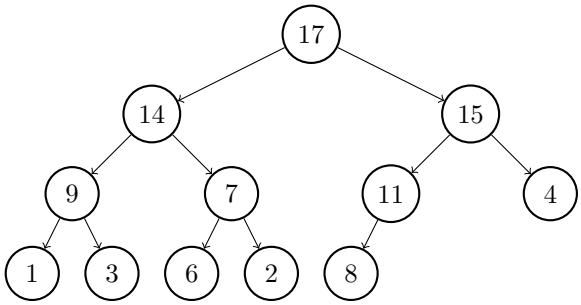
0	1	2	3	4	5	6	7	8	9	10	···
17	14	11	9	7	8	4	1	3	6	2	nil

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

Für min Heaps muss lediglich das Ungleichheitszeichen umgedreht werden.



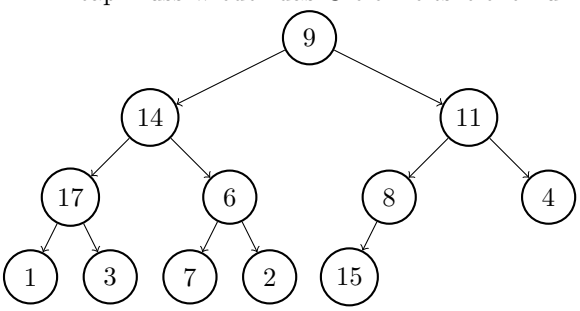
Insert 15, before fixup



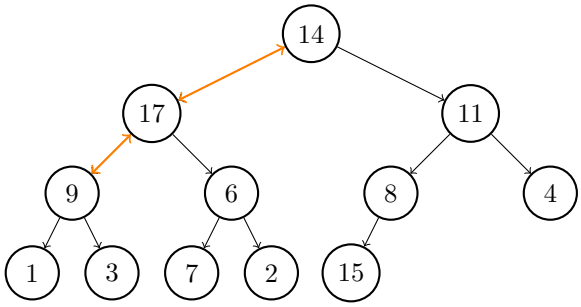
After fixup

```
    void heapify(int i) { // O(h) = O(log n)
        // Used if heap is new unsorted array.
        // If heap was build using insert, this should not be necessary
        int max = i;
        int l = left(i);
        int r = right(i);
        if(l < size && heap[i] < heap[l]) // If left child is bigger than i
            max = l; // Set max to left
        if(r < size && heap[max] < heap[r]) // if right child is bigger than max (i or left)
            max = r; // Set max to right
        if(max != i) { // If max is not i -> max is left or right
            int temp = heap[i];
            heap[i] = heap[max];
            heap[max] = temp;
            // Swap i and max
            heapify(max); // max is now i
            // Move down the tree
        }
    }
```

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.

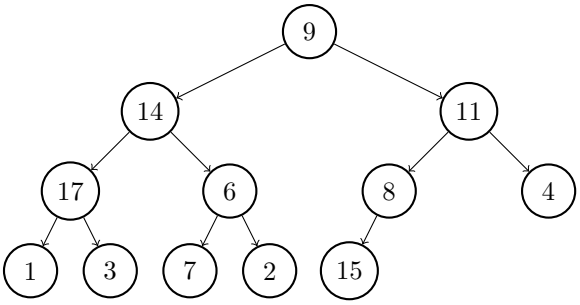


Before

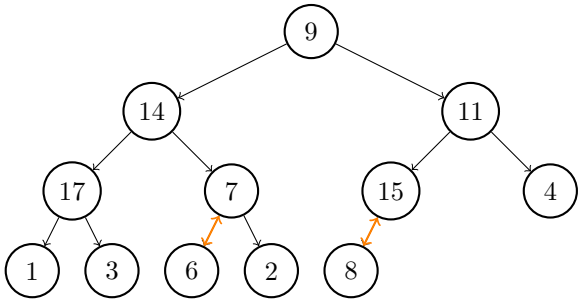


Heapify root

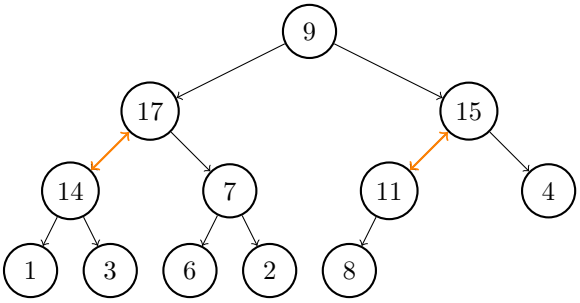
```
    void buildHeap() { // O(nh) = O(n log n)
        for(int i = parent(size - 1); i >= 0; i--)
            heapify(i); // Calls heapify on each node starting from the second to last row
        // Heap should now be sorted accordingly -> biggest node at root
    }
    int[] heapSort() { // O(nh) = O(n log n)
        // Assumes new array for heap
        buildHeap(); // Sorts heap accordingly
        int[] res = new int[size]; // Create new array for sorted result
        int i = 0;
        while(!isEmpty())
            res[i++] = extractMax(); // Extract max and add to array
        return res; // Array is now sorted in reverse natural order
    }
    int extractMax() { // O(h) = O(log n)
        if(isEmpty())
            throw new UException("Underflow");
        int max = heap[0]; // Biggest value at root
        heap[0] = heap[size - 1]; // Sets root to last element
        size--; // Decrease size
        heapify(0); // Heapify new root
        return max;
    }
}
```



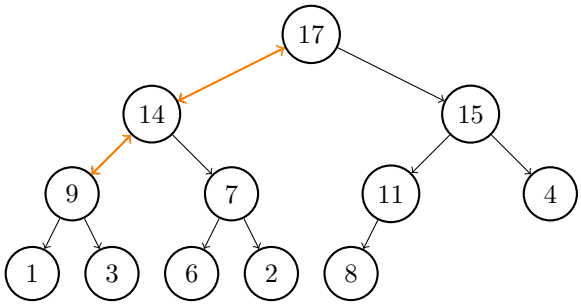
Before heapsort



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

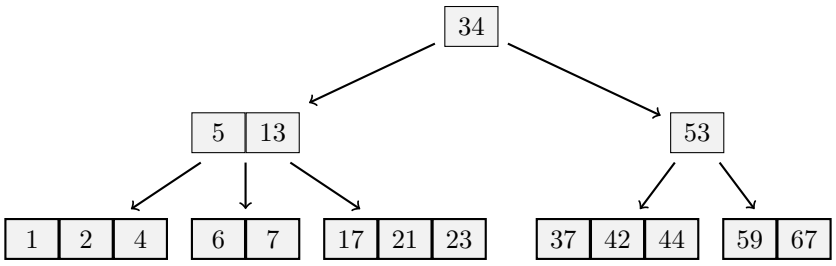
6.5 B-Tree

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
  $\implies$ für alle Werte k_j in j-ten Kind gilt: $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \ldots \leq key[m-1] \leq k_m$

Ein B-Tree kann somit in einem Knoten mehr als einen Wert und mehr als zwei Kinder besitzen. B-Trees balanzieren 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.

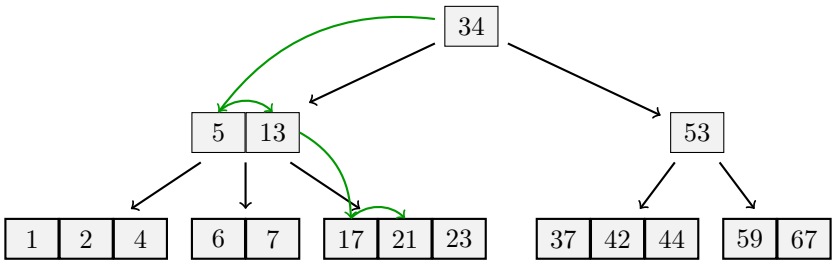
```java
class BTree {
    class BNode {
        int[] keys; // array of all keys in the node
        int t; // degree, defines the maximum number of keys in the node
        BNode[] children; // array of children to the node
        int n; // current number of keys in the node, used to find the first free index
        boolean isLeaf; // true if the node is a leaf node -> no children
    }
    BNode root;
    final int t;
    BTree(int t) {
        root = null;
        this.t = t;
    }
```



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

```java
class BTree {
    class BNode {
        int findKey(int k) { // Finds index of key in node, i = n if not in node
            int i = 0;
            while (i < n && keys[i] < k) i++;
            return i;
        }
        boolean inNode(int i) {
            return (i < n && keys[i] == 0);
        }
        BNode search(int k) {
            int i = findKey(k); // find key
            if (inNode(i)) // If k in node
                return this;
            if (isLeaf) // If k not in node and node is leaf, k doesnt exist in tree
                return null;
            return children[i].search(k); // search for k in corresponding child
        }
    }
    void search (int k) {
        if (root == null) {
            System.out.println("Tree is empty");
            return;
        }
        root.search(k); // Search for k
    }
}
```

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.

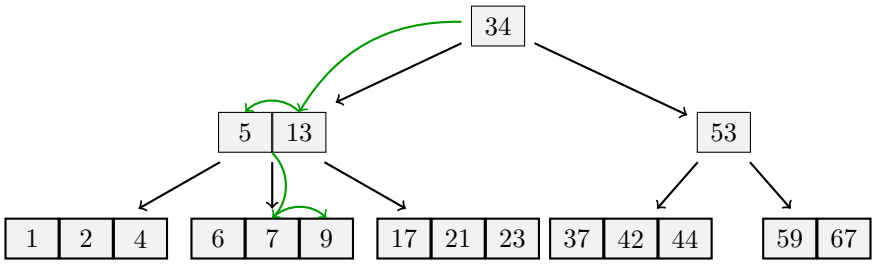


Search 21

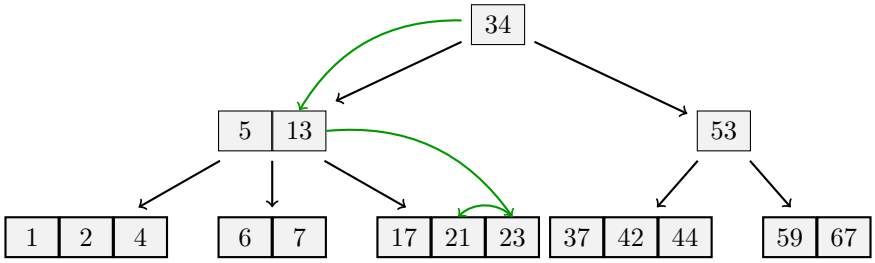
```
class BTree {
    class BNode {
        int findKey(int k) { // Finds index of key in node, i = n if not in node
            int i = 0;
            while (i < n && keys[i] < k) i++;
            return i;
        }
        boolean inNode(int i) {
            return (i < n && keys[i] == 0);
        }
        void insertNonFull(int k) { // Omega(1), O(t), Theta(t)
            int i = n -1; // first free index
            if (isLeaf) { // If node has no children
                while (i >= 0 && keys[i] > k) {
                    keys[i + 1] = keys[i];
                    i--;
                } // Moves through the array and moves elements bigger than k to the right
                // Creates insertion point
                keys[i + 1] = k; // Inserts k
                n++; // Increases number of keys
            } else { // If node has children
                while (i >= 0 && keys[i] > k) i--; // Moves through array and finds insertion point
                if (children[i + 1].n == 2 * t - 1) { // If insertion child is full
                    splitChild(i + 1, children[i + 1]); // Splits insertion child
                    if (keys[i + 1] < k) // If key at insertion point is smaller than k
                        i++; // Move insertion point one to the right
                }
                children[i + 1].insertNonFull(k); // Insert k into insertion child
            }
        }
        void splitChild(int i, BNode y) {
            BNode z = new BNode(y.t, y.isLeaf); // Creates new node akin to y
            z.n = t - 1; // Has half the number of keys as the node to be split
            for (int j = 0; j < t - 1; j++) // Copies the second half of the keys to the new node
                z.keys[j] = y.keys[j + t];
            if (!y.isLeaf) { // If y has children
                for (int j = 0; j < t; j++) // Copies the second half of the children to the new node
                    z.children[j] = y.children[j + t];
            }
            y.n = t - 1; // Node now has half the keys it had before
            for (int j = n; j > i; j--) // Searches for insertion point of new child
                children[j + 1] = children[j];
            children[i + 1] = z; // Inserts new child
            for (int j = n - 1; j >= i; j--) // Creates insertion point for new key
                keys[j + 1] = keys[j];
            keys[i] = y.keys[t - 1]; // Inserts new key
            n++; // Increases number of keys
        }
    }
    void insert(int k) {
        if (root == null) {// If tree is empty
            root = new BNode(t, true); // Create root
            root.keys[0] = k; // Insert k
            root.n = 1; // Increase number of keys
        } else { // If tree is not empty
            if (root.n == 2 * t - 1) { // If root is full
                BNode s = new BNode(t, false); // Create new node
                s.children[0] = root; // set root as first child of new node
                s.splitChild(0, root); // Split root
                int i = 0;
                if (s.keys[0] < k) i++; // If only key in new node is smaller than k, move insertion
    point
                s.children[i].insertNonFull(k); // Insert k in corresponding child
                root = s; // sets new node as root
            } else // If root is not full
                root.insertNonFull(k); // Simply insert
        }
    }
}
```

Prinzipiell folgt dieser Algorithmus einfach der Folge:

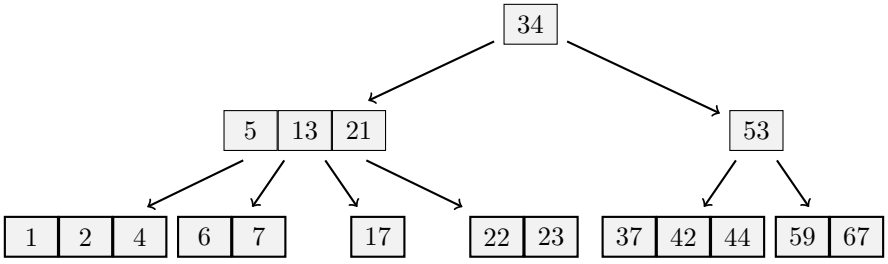
1. Finde Einfügepunkt
2. Wenn Node schon $2 \cdot t - 1$ Werte besitzt, splitte es
   - Teile Node in zwei Nodes mit je $t - 1$ Werten
   - Der mittlere Knoten wird in den Elternknoten eingefügt
   - Wenn dadurch der Elternknoten $2 \cdot t - 1$ Werte besitzt, splitte diesen rekursiv nach oben
3. Wert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



Insert 9 (Simple Case)



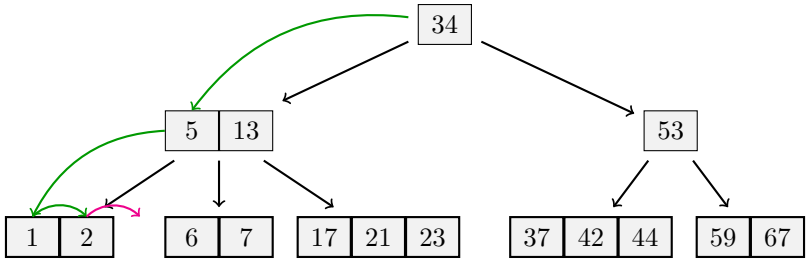
Before Insert 22 (Needs splitting)



Insert 22

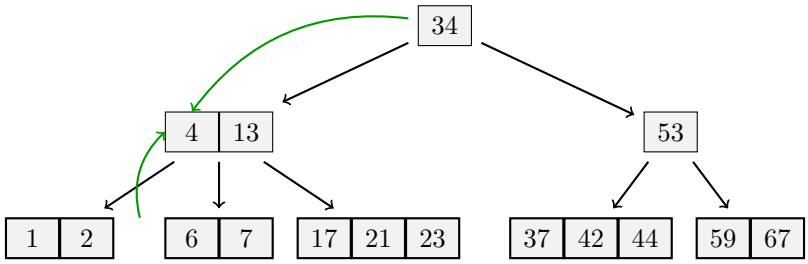
```
class BTree {
    class BNode {
        int findKey(int k) { // Finds index of key in node, i = n if not in node
            int i = 0;
            while (i < n && keys[i] < k) i++;
            return i;
        }
        boolean inNode(int i) {
            return (i < n && keys[i] == 0);
        }
        int getPredecessor(int i) {
            BNode child = children[i];
            while (!child.isLeaf)
                child = child.children[child.n];
            return child.keys[child.n - 1];
        }
        int getSuccessor(int i) {
            BNode child = children[i + 1];
            while (!child.isLeaf)
                child = child.children[0];
            return child.keys[0];
        }
        void fill(int i) {
            if (i != 0 && children[i - 1].n >= t) // If the previous child is filled more than half
                borrowFromPrev(i);
            else if (i != n && children[i + 1].n >= t) // If the next child is filled more than half
                borrowFromNext(i);
            else // If both children are not filled more than half
                if (i != n) // Merge with next child
                    merge(i);
                else // Merge with previous child
                    merge(i - 1);
        }
        void borrowFromPrev(int i) {
            BNode child = children[i];
            BNode sibling = children[i - 1]; // Get previous child, to be borrowed from
            for (int j = child.n - 1; j >= 0; j--)// Move keys to the right
                child.keys[j + 1] = child.keys[j];

            if (!child.isLeaf)
                for (int j = child.n; j >= 0; j--) // Move children to the right
                    child.children[j + 1] = child.children[j];
            child.keys[0] = keys[i - 1]; // Move key from parent to child
            if (!child.isLeaf) // Move children if not child not a leaf
                child.children[0] = sibling.children[sibling.n];
            keys[i - 1] = sibling.keys[sibling.n - 1]; // Move key from sibling to parent
            child.n++; // Increase number of keys in child
            sibling.n--; // Decrease number of keys in sibling
        }
        void borrowFromNext(int i) {
            BNode child = children[i];
            BNode sibling = children[i + 1]; // Get next child, to be borrowed from
            child.keys[child.n] = keys[i]; // Move key from parent to child
            if (!child.isLeaf) // if child isnt a leaf move last first child of sibling to child
                child.children[child.n + 1] = sibling.children[0];
            keys[i] = sibling.keys[0]; // Move key from sibling to parent
            for (int j = 1; j < sibling.n; j++)// Move keys to the left
                sibling.keys[j - 1] = sibling.keys[j];
            if (!sibling.isLeaf)
                for (int j = 1; j <= sibling.n; j++)// Move children to the left
                    sibling.children[j - 1] = sibling.children[j];
            child.n++; // Increase number of keys in child
            sibling.n--; // Decrease number of keys in sibling
        }
        void merge(int i) {
            BNode child = children[i];
            BNode sibling = children[i + 1];
            child.keys[t - 1] = keys[i]; // Move key from parent to child
            for (int j = 0; j < sibling.n; j++) // Move keys from sibling to second half of child
                child.keys[j + t] = sibling.keys[j];
```

```
            if (!child.isLeaf)
                for (int j = 0; j <= sibling.n; j++) // Move children from sibling to second half of child
                    child.children[j + t] = sibling.children[j];
            for (int j = i + 1; j < n; j++) // Move keys to fill the gap created by moving to child
                keys[j - 1] = keys[j];
            for (int j = i + 2; j <= n; j--) // Move children to fill the gap created by moving to child
                children[j - 1] = children[j];
            child.n += sibling.n + 1;
            n--;
        }
        void delete(int k) {
            int i = findKey(k);
            if (inNode(i)) {
                if (isLeaf)
                    deleteFromLeaf(i);
                else
                    deleteFromNonLeaf(i);
            } else {
                if (isLeaf) {
                    System.out.println("Key not found");
                    return;
                }
                boolean flag = (i == n); // if key is present in subtree of last child
                if (children[i].n < t) // If child that contains key has less than t keys
                    fill(i); // Fill that child
                if (flag && i > n) // If key is present in subtree of last child
                    children[i - 1].delete(k); // Delete from that subtree
                else // If key is not present in subtree of last child
                    children[i].delete(k); // Delete from that subtree
            }
        }
        void deleteFromLeaf (int i) {
            for (int j = i + 1; j < n; j++) // Move all keys after i to the left
                keys[j - 1] = keys[j];
            n--; // reduce number of keys
        }
        void deleteFromNonLeaf (int i) {
            int k = keys[i];
            if (children[i].n >= t) { // If child that contains key has more than t keys
                int pred = getPredecessor(i); // Get predecessor
                keys[i] = pred; // Replace key with predecessor
                children[i].delete(pred); // Delete predecessor from child
            } else if (children[i + 1].n >= t) {// If child that contains key has more than t keys
                int succ = getSuccessor(i); // Get successor
                keys[i] = succ; // Replace key with successor
                children[i + 1].delete(succ); // Delete successor from child
            } else { // If both children have less than t keys
                merge(i); // Merge children
                children[i].delete(k); // Delete key from child
            }
        }
    }
    void delete(int k) {
        if (root == null) {
            System.out.println("Tree is empty");
            return;
        }
        root.delete(k); // Delete k
        if (root.n == 0) { // If root is empty
            if (root.isLeaf) // If root is leaf -> tree is empty
                root = null; // Delete root
            else
                root = root.children[0]; // Replace root with its child
        }
    }
}
```



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 unnötioge 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

7.1 Deterministisch und Probabilistisch

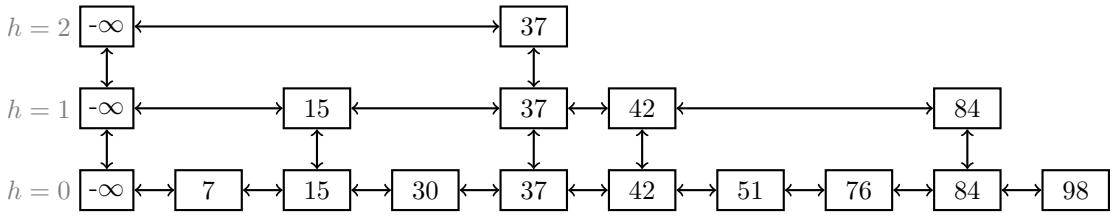
Bisher waren alle Datenstrukturen deterministisch, d.h., dass für die selben Eingaben das Verhalten immer gleich sein wird.
Bei probabilistischen Datenstrukturen ist das Verhalten nicht nur von Eingaben abhängig, sondern auch von zufälligen Faktoren.

Aspekt	Deterministische Datenstrukturen	Probabilistische (Randomisierte) Datenstrukturen
Vorteile		
Leistungsgarantien	Bietet garantierte Worst-Case-Zeitkomplexität.	Bietet gute durchschnittliche Leistung, oft schneller in der Praxis.
Vorhersehbarkeit	Verhalten ist für die gleichen Eingaben vorhersehbar und konsistent.	Flexibler und vermeidet Worst-Case-Szenarien unter typischen Bedingungen.
Worst-Case-Behandlung	Speziell entwickelt, um Worst-Case-Szenarien zu handhaben.	Vermeidet Worst-Case-Szenarien durch probabilistische Methoden.
Einfachheit	Konzeptuell einfach mit klaren Regeln (z.B. AVL-Baum-Rotationen).	Oft einfacher in der Implementierung, ohne komplexe Ausgleichsoperationen.
Stabilität	Deterministisches Verhalten führt zu stabilen und wiederholbaren Ergebnissen.	Flexibel und widerstandsfähig gegenüber unterschiedlichen Eingabemustern.
Nachteile		
Komplexität in der Implementierung	Erfordert oft komplexe Ausgleichslogik (z.B. Rot-Schwarz-Bäume, AVL-Bäume).	Einfacher, aber schwerer probabilistisch zu analysieren.
Speicherbedarf	Kann zusätzlichen Speicher für die Speicherung von Ausgleichsinformationen erfordern.	Kann speichereffizient sein, aber einige Strukturen (z.B. Bloom-Filter) können eine geringe Fehlerquote aufweisen.
Handhabung spezifischer Eingaben	Kann bei bestimmten Eingabemustern schlecht abschneiden (z.B. Quicksort mit sortierten Eingaben).	Vermeidet schlechte Leistung bei bestimmten Eingaben durch Zufälligkeit.
Vorhersehbarkeit	Vorhersehbar und kann von einem Gegner ausgenutzt werden (z.B. Hash-Kollisionsangriffe).	Weniger vorhersehbar, wodurch die Wahrscheinlichkeit sinkt, dass gegnerische Eingaben zu einer Worst-Case-Leistung führen.
Komplexität in der Analyse	Einfacher zu analysieren und zu verstehen in Bezug auf Worst-Case-Verhalten.	Erfordert probabilistische Analyse, um Leistungsgarantien zu verstehen.
Wesentliche Unterschiede		
Leistungsgarantien	Strikte Worst-Case-Leistungsgarantien.	Konzentriert sich auf erwartete durchschnittliche Leistung.
Verhalten unter adversen Bedingungen	Kann unter adversen Bedingungen erheblich nachlassen (z.B. bestimmte Hashing-Methoden).	Robuster gegenüber adversen Bedingungen aufgrund von Zufälligkeit.
Implementierungskomplexität	Kann mehr Aufwand erfordern, um eine optimale Leistung zu gewährleisten (z.B. Ausbalancierung von Bäumen).	Typischerweise einfacher zu implementieren mit guter durchschnittlicher Leistung.
Anwendungsfälle	Geeignet für Anwendungen, bei denen Worst-Case-Leistung entscheidend ist.	Ideal für Anwendungen, bei denen die durchschnittliche Leistung wichtiger ist.

7.2 Skip-Lists

Skip-Lists sind eine Datenstruktur, die einer Linked-List sehr ähnelt. Sie baut auf der selben grundlegenden Struktur auf, erweitert diese jedoch noch zusätzlich. So haben Elemente in einer Skip-List nicht nur `next` (und eventuell `prev`) sondern auch noch `up` und `down`.
Die Struktur einer Skip-List ähnelt sogesehen mehreren aufeinandergestapelten Linked-Lists. Beim Aufbau der Skip-List wird für jedes Element zufällig ausgesucht, auf wie vielen Ebenen es abgebildet ist. Dies ergibt den Vorteil, dass diese Struktur beim `insert`, `delete` und `search` eine average time complexity von $O(\log n)$ besitzt, was sie mit balanzierten Bäumen (AVLT, RBT...) vergleichbar macht. Der Vorteil von Skip-Lists über Bäume sind unter anderem, dass sie einfacherer zu implementieren sind und weniger Speicher brauchen. Die Nachteile bilden hierbei die schlechte Worst-Case Performance von $O(n)$.
Skip-Lists werden oft in Datenbanken genutzt um Daten nach einer spezifierten Ordnung zu speichern, aber auch für Datensätze, die oft modifiziert werden müssen, da das anwenden von anderen Algorithmen auf Skip-Lists oft relativ einfach ist.

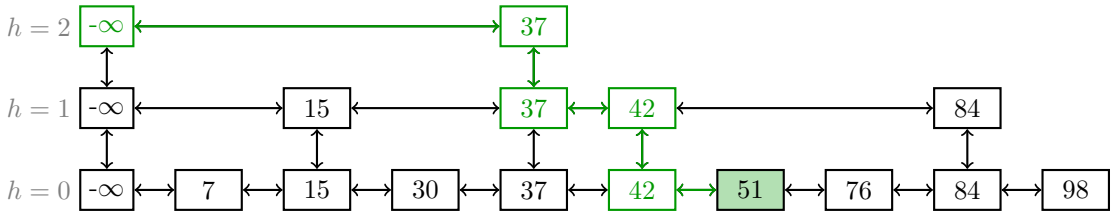
```
class SkipList {
    class SkipNode {
        int key;
        SkipNode next;
        SkipNode prev;
        SkipNode down;
        SkipNode up;
        SkipNode(int k) {
            key = k;
        }
    }
    SkipNode head; // First node in highest level
    int height; // starts at 0 -> biggest list at height = 0
    final double P = 0.5; // Probability of inserted node being added to express list
    SkipList() {
        head = new SkipNode(Integer.MIN_VALUE);
        height = 0;
    }
```



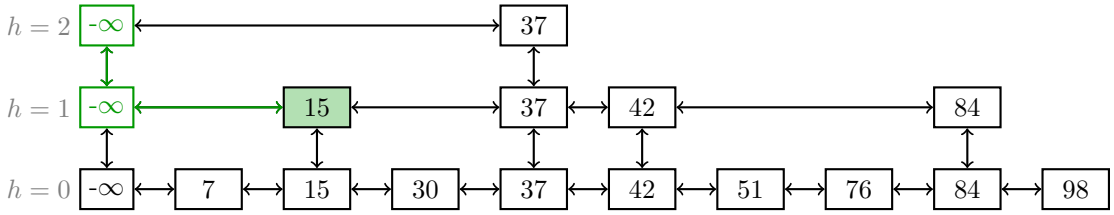
Example

Der Suchalgorithmus ist relativ simpel. Er startet beim Head (Höchster Knoten des Starts) und durchläuft jedes Level in order, bis der nächste Knoten größer ist als der gesuchte Knoten / `null` ist. Dann geht er an dem Knoten nach unten und durchsucht diese Liste weiter. Dies geschieht, bis er entweder den Wert gefunden hat, in welchem Fall er den gefundenen Knoten zurück gibt, oder bis er null erreicht, was bedeutet, dass der Wert nicht in der Liste ist.

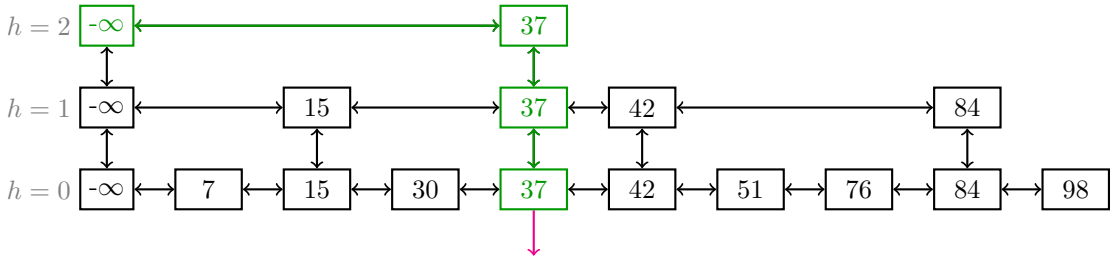
```
1     SkipNode search(int k) {
2         SkipNode curr = head;
3         while (curr != null) {
4             if (curr.key == k) // key found
5                 return curr;
6             else if (curr.next != null && curr.next.key <= k) // If next key is less than or equal to k
7                 curr = curr.next; // move along the list
8             else // If next key is greater than k
9                 curr = curr.down; // move down the list
10        }
11        return null; // key not found
12    }
```



Search 51



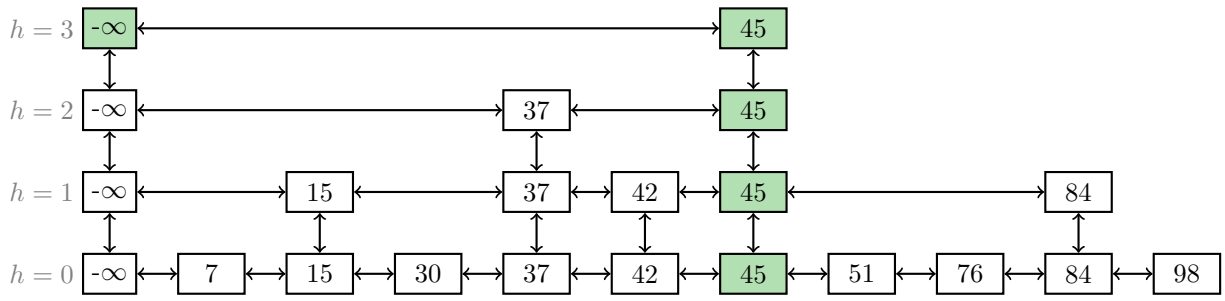
Search 15



Search 40

Einfügen in die Skip-List ist ein wenig komplizierter, aber auch grundlegend simpel. Erst einmal muss ermittelt werden auf wie viele Ebenen der Wert hinzugefügt werden muss. Dies kann über eine vom Element abhängige Formel oder auch andere Methoden zum erzeugen einer zufälligen Zahl geschehen. Nachdem dieses Level ermittelt wurde, muss, falls notwendig die Liste um die noch nicht vorhandenen Ebenen ergänzt werden. Dies geschieht dadurch, dass man die `Head` Node nach oben erweitert, den `height`-Counter erhöht und den Head auf den neuen Head aktualisiert. Nachdem dies geschehen ist wird nun die Liste wie beim `search`-Algorithmus durchgangen um die Einfügestelle zu ermitteln. Beim Durchlauf werden zusätzlich noch die zukünftigen Vorgängerknoten in einem Array vermerkt. Nachdem werden nun die Knoten auf den bestimmten Ebenen eingefügt.

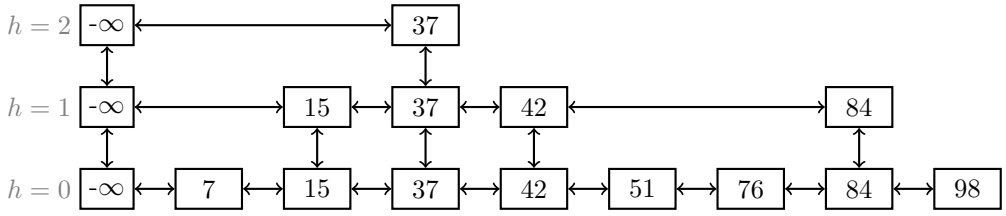
```
    int randomLevel() { // Used to determine in how many lists a node will be added
        double r = Math.random();
        int lvl = 0;
        while (r < P) {
            lvl++;
            r = Math.random();
        }
        return lvl;
    }
    void insert(int k) {
        int lvl = randomLevel();
        while (lvl > height) { // If needed increase list height and add required heads
            head.up = new SkipNode(Integer.MIN_VALUE);
            head.up.down = head;
            head = head.up;
            height++;
        }
        SkipNode[] prevs = new SkipNode[height + 1]; // Holds the previous nodes in each list
        SkipNode curr = head;
        for (int i = height; i >= 0; i--) { // For each level starting from the highest
            while (curr.next != null && curr.next.key < k)
                // Loops through current list until next key is greater than or equal to k
                curr = curr.next;
            if (curr.key == k) return; // key already in list
            prevs[i] = curr; // Adds current node as a predecessor to the new node
            curr = curr.down; // Moves down in the list
        }
        int count = 0; // counter for number of lists in which the new node has been added
        SkipNode dwn = null; // Holds the node that is the down node of the node in a level
        while (count <= lvl) { // Add new nodes to lists in lvl
            SkipNode newNode = new SkipNode(k);
            newNode.next = prevs[count].next; // Includes the new node in the list
            newNode.prev = prevs[count];
            if (prevs[count].next != null) // If there is a next node
                prevs[count].next.prev = newNode; // change previous to new node
            prevs[count].next = newNode; // change next to new node
            newNode.down = dwn; // connect newNode to itself on a lower level
            if (dwn != null) // If added to more than 1 level
                dwn.up = newNode; // connect lower level to newNode
            dwn = newNode; // Update dwn
            count++; // Update counter
        }
    }
```



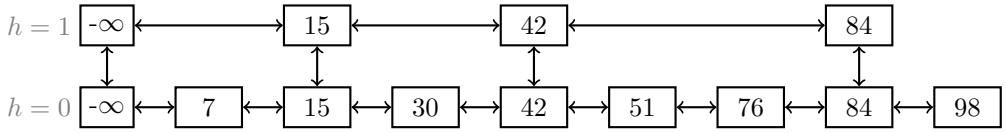
Insert 45 with randomLevel = 3

Delete ist wieder relativ simpel. Es wird erstmal die zu löschende Node gesucht. Nachdem muss lediglich die Referenzen von den Vor- und Nachgängern abgeändert werden, sodass das Element selbst nichtmehr in der Liste ist. Dies muss nun lediglich nur für jede Ebene wiederholt werden.

```
    void delete(int k) {
        SkipNode node = search(k); // Find first node in list that has k
        while (node != null) { // Moves down from the found node
            // Remove node from list by removing its references
            node.prev.next = node.next;
            if (node.next != null)
                node.next.prev = node.prev;
            if (node.next == null
                    && node.prev.key == Integer.MIN_VALUE
                    && node.prev.down != null) {
                // If deleted node is the last node in the list, reduce list height and update head
                node.prev.down.up = null;
                head = node.prev.down;
                height--;
            }
            node = node.down;
        }
    }
}
```



Delete 30



Delete 37

7.3 Hash Tables

7.4 Bloom Filter

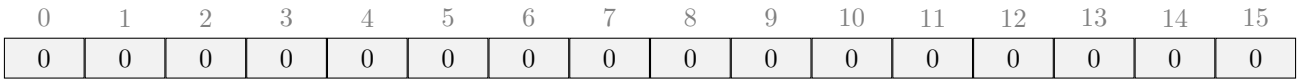
Ein Bloom-Filter ist eine Datenstruktur, die benutzt wird um schnell herauszufinden, ob ein Element in einer Datenstruktur vorkommt. Sie ist ideal für große Datensätze, bei denen *false-positives* akzeptabel sind, *false-negatives* nicht. D.h. sie können zuverlässig sagen, ob ein Element vorkommt, können aber auch anschlagen, wenn ein Element nicht explizit eingefügt wurde.
Bloom Filter sind sehr effizient, da sie zum einem nicht die Elemente selbst speichern, sondern nur ihre Anwesenheit, zum anderen, da das Einfügen und Auslesen auch sehr schnell ist mit $O(k)$ mit k die Anzahl der Funktionen.
Nachteile von Bloom Filtern sind die *false-positives*, die sehr komplizierte deletion (Ein Element rauslöschen kann auch anderes rauslöschen, was *false-negatives* erzeugt) und die festgelegte Größe.
Sie werden häufig genutzt um Caches und Spam zu filtern, aber auch um bspw. zu schauen, ob ein Passwort häufig verwendet wird.

```java
import java.util.function.Function;
class BloomFilter {
    Function<String, Integer>[] functions;
    boolean[] bloomFilter;
    int bloomSize;
    BloomFilter(int bloomSize, Function<String, Integer>[] functions) {
        this.functions = functions; // functions that will be used
        this.bloomSize = bloomSize; // size of the bloom filter
        bloomFilter = new boolean[bloomSize]; // defaults to false
    }
    void exampleFunctions() {
        functions = new Function[3];
        functions[0] = (String s) -> s.length() % bloomSize;
        functions[1] = (String s) -> s.charAt(0) % bloomSize;
        functions[2] = (String s) -> s.charAt(s.length() - 1) % bloomSize;
    }
```

Bei diesen Beispielfunktionen, könnte man z.B. **Dance** einfügen, was aber die gleichen bits wie **Dodge** belegt. Demnach würde bei der Suche für **Dodge** ein *false-positive* erzeugt werden. False-Positives werden immer wahrscheinlicher je mehr Elemente eingefügt werden und je kleiner der Bloom Filter ist. Natürlich sind sie aber auch grundlegend von den Functions, bezüglich der Komplexität, Probabilistik und Anzahl, abhängig. Diese Beispielfunktionen sind relativ schlecht, da sie simpel sind, was im Endeffekt in mehr *false-positives* resultiert. Abstraktere, komplexere Funktionen funktionieren hierbei meist besser, da sie nicht an Adjezenz der Eingaben festhalten.

Einfügen ist sehr simpel. Es müssen lediglich die Funktionen auf die Eingabe angewendet werden und die entsprechenden Bits danach umgestellt werden.

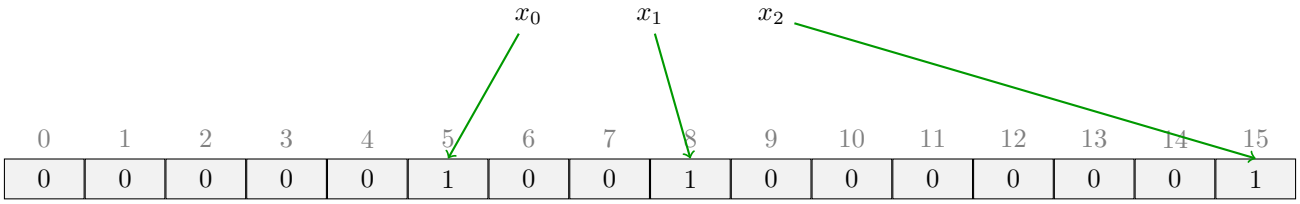
```
1    void addToBloom(String x) { // O(k), k = number of functions
2        for (Function<String, Integer> function : functions) { // for each function
3            bloomFilter[function.apply(x)] = true; // add to bloom
4        }
5    }
```



Before (bloomSize = 16)

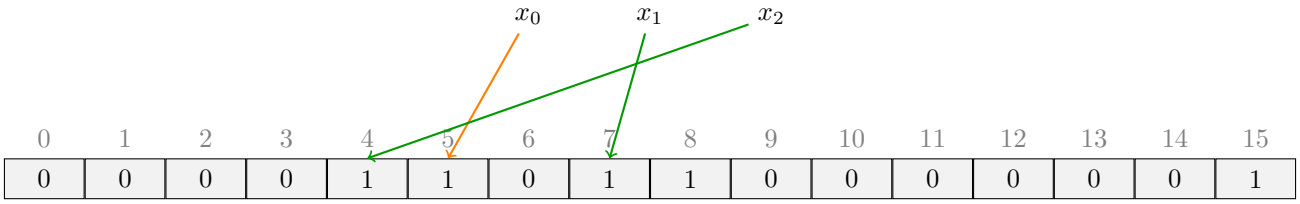
Insert "hello":

- $x_0 = \text{"}hello\text{"}.length() \,\%\, 16 = 5$
- $x_1 = (int)\,'h' = 104 \,\%\, 16 = 8$
- $x_2 = (int)\,'o' = 111 \,\%\, 16 = 15$



Insert "world":

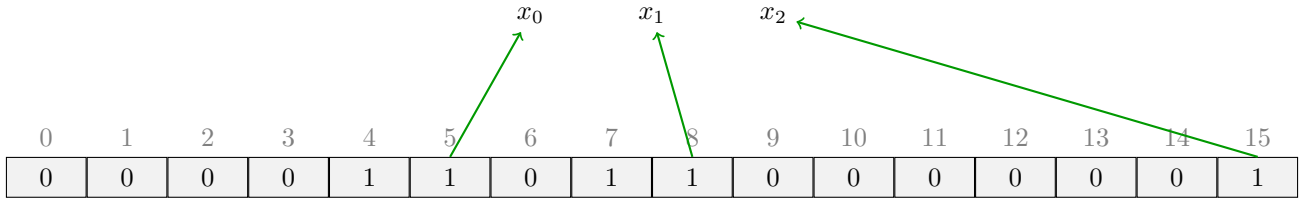
- $x_0 = \text{"}world\text{"}.length() \,\%\, 16 = 5$
- $x_1 = (int)\,'w' = 119 \,\%\, 16 = 7$
- $x_2 = (int)\,'d' = 100 \,\%\, 16 = 4$



Das Durchsuchen des Bloom-Filters läuft fast identisch zum Einfügen ab. Es müssen wieder mittels der Funktionen die passenden Werte der Eingabe gefunden werden. Diese werden dann verwendet um zu schauen, ob alle benötigten Bits im Bloom-Filter vorhanden sind. Wenn alle da sind ist das Element im Bloom-Filter enthalten, allerdings kann dies auch wahr für Werte sein, die nicht explizit eingefügt wurden (false-positive).

```
    boolean isInBloom(String x) { // O(k), k = number of functions
        for (Function<String, Integer> function : functions) { // for each function
            if (!bloomFilter[function.apply(x)]) // if not in bloom
                return false; // One function is not in bloom -> false
        }
        return true; // If all functions are present in bloom -> true
    }
}
```

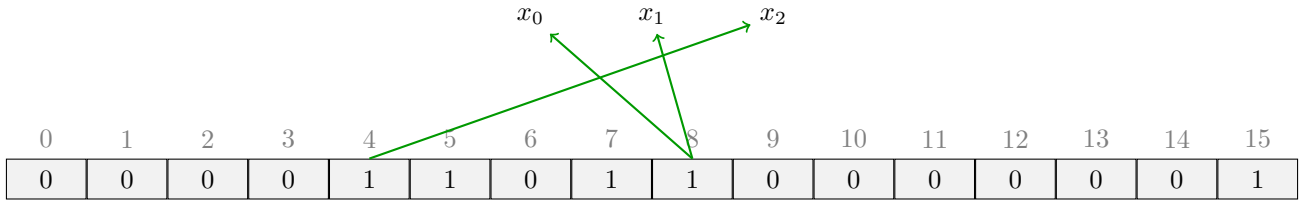
Check if "hello" exists: (true-positive)



$\Longrightarrow$ Existenz im Bloom Filter

Check if "harassed" exists: (false-positive)

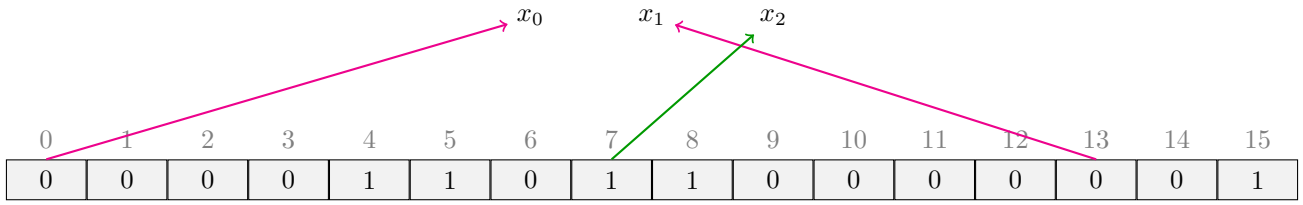
- $x_0 = \text{"}harassed\text{"}.length() \,\%\, 16 = 8$
- $x_1 = (int)\,'h' = 104 \,\%\, 16 = 8$
- $x_2 = (int)\,'d' = 100 \,\%\, 16 = 4$



$\Longrightarrow$ Existenz im Bloom Filter obwohl nicht spezifisch eingefügt

Check if "misunderstanding" exists: (true-negative)

- $x_0 = \text{"}misunderstanding\text{"}.length() \,\%\, 16 = 0$
- $x_1 = (int)\,'m' = 109 \,\%\, 16 = 13$
- $x_2 = (int)\,'g' = 103 \,\%\, 16 = 7$



$\Longrightarrow$ keine Existenz im Bloom Filter

8 Graphen Algorithmen