AuD - Zusammenfassung

Moritz Gerhardt

1 Inhaltsverzeichnis

2 Was ist ein Algorithmus?

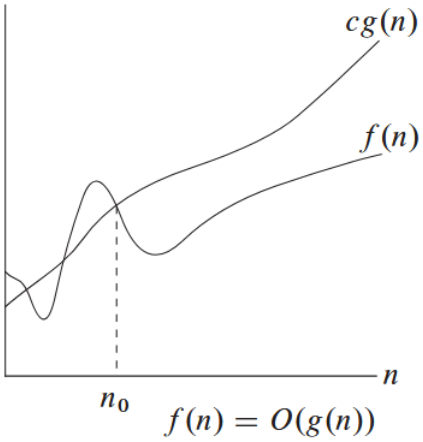
Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe.
Dabei sollte ein Algorithmus im allgemeinen folgende Vorraussetzungen erfüllen:

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:**

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

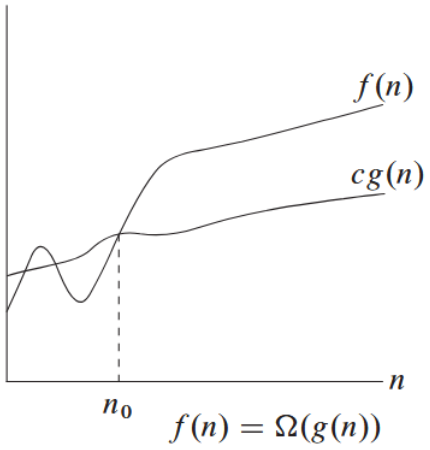
  Die Komplexität der Summe zweier Funktionen ist der Maximalwert der Komplexität der beiden Funktionen.
- **Multiplikation:**

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

  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:**

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

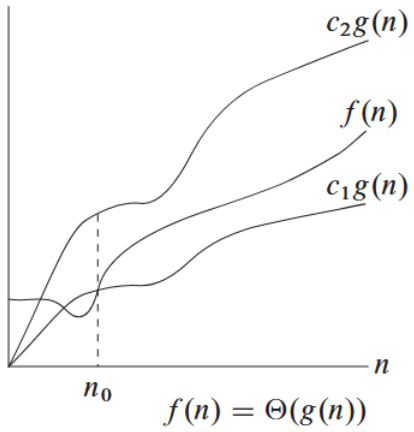
  Die Komplexität der Summe zweier Funktionen ist der Minimalwert der Komplexität der beiden Funktionen.
- **Multiplikation:**

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

  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)

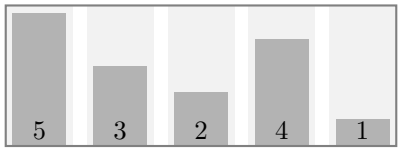
4.2 Insertion Sort

```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
        }
    }
}
```

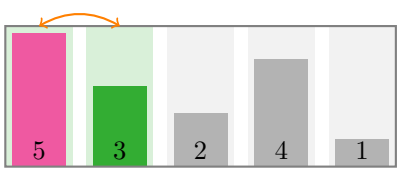
4.2 (a) Vorgehensweise

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.

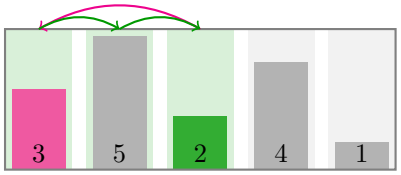
4.2 (b) Visuelle Darstellung



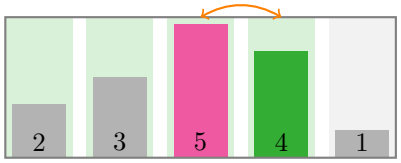
Anfangszustand



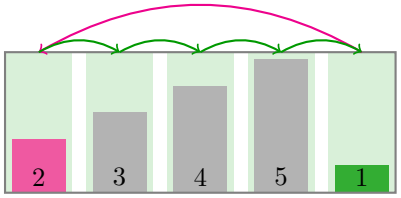
1. Iteration



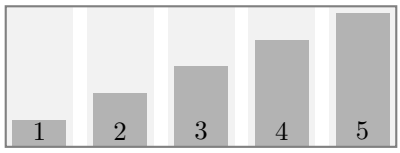
2. Iteration



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 array1, der in reverse order sortiert ist.
  - Demnach muss jedes Element den kompletten array1 durchlaufen.
  - Dies ergibt eine Worst-Case Laufzeit von $\Theta(n^2)$
- **Best-Case:**
  - Der Best-Case ist ein array1, der schon sortiert ist.
  - Demnach muss kein Element verschoben werden, aber trotzdem muss bei jedem Element einmal geprüft werden, ob es größer als sein Vorgänger ist.
  - Dies ergibt eine Best-Case Laufzeit von $\Theta(n)$
- **Average-Case:**
  - Der Average-Case ist ein array1, der in random order sortiert ist.
  - Demnach muss für jedes Element der array1 durchschnittlich bis zur Hälfte durchlaufen werden.
  - Nach der quadratischen Steigerung für große Zahlen ist die Hälfte aber irrelevant, weswegen $\Theta(n^2)$ ist.

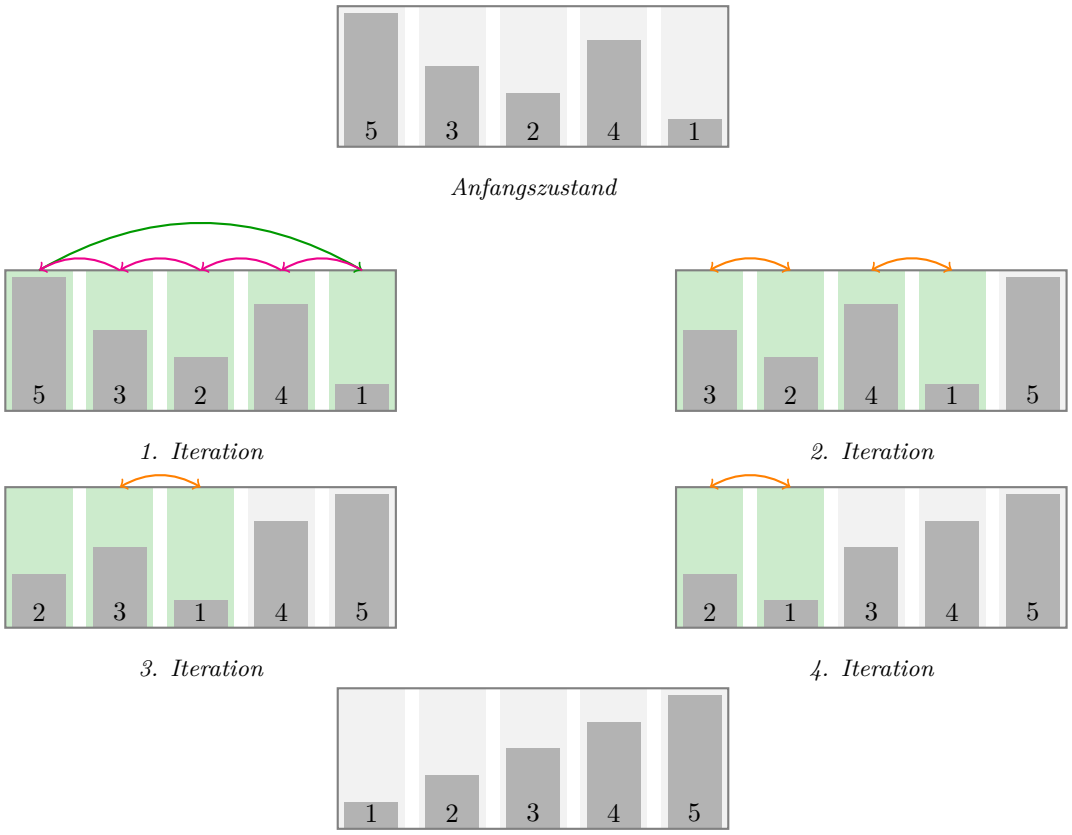
4.3 Bubble Sort

```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 $\Theta(n^2)$

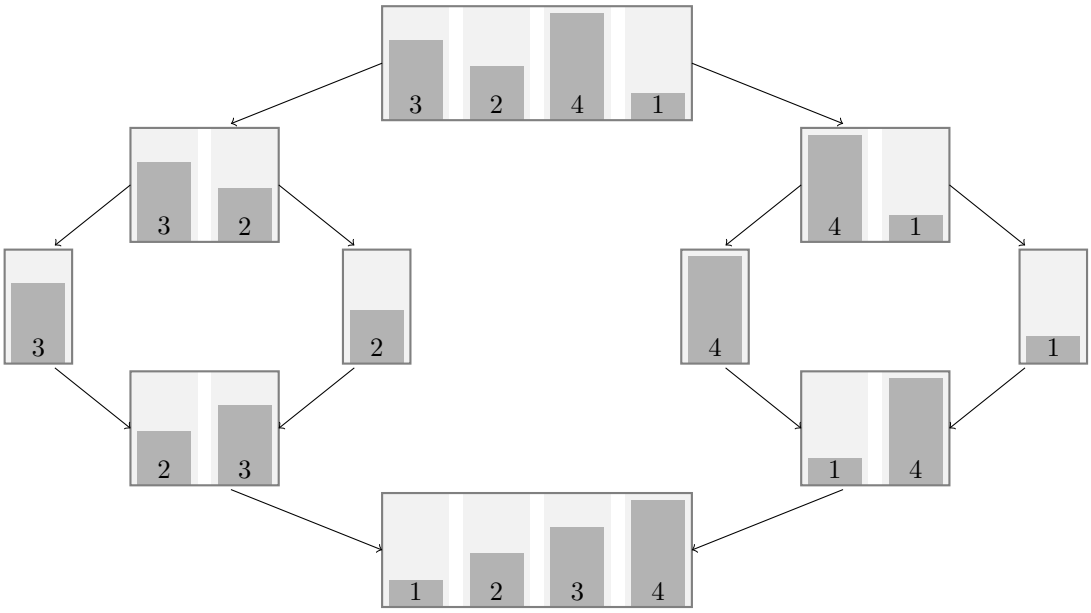
4.4 Merge Sort

```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

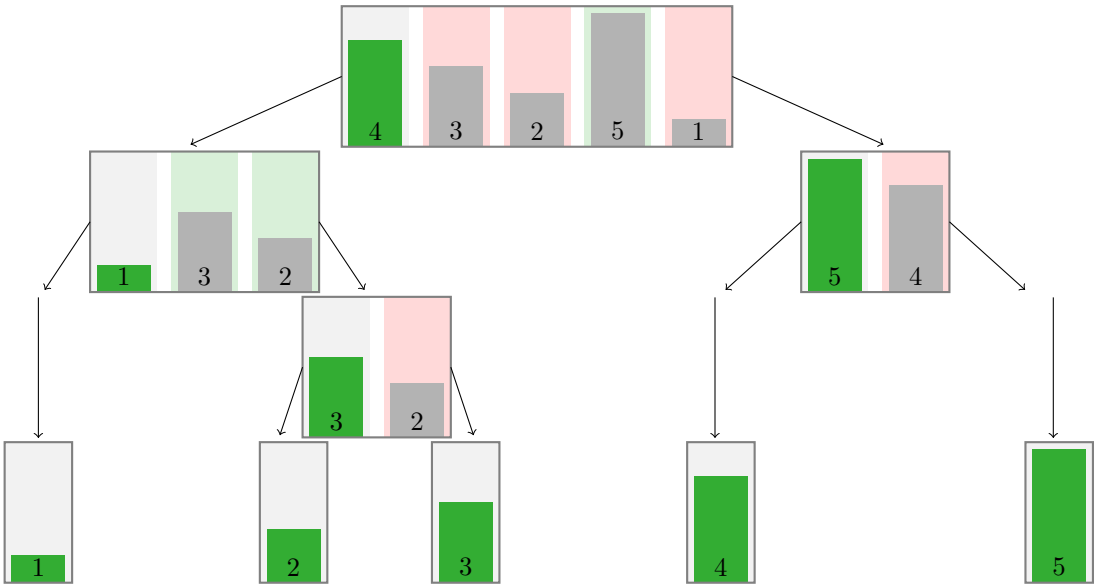
- **Worst-Case:**
  - Der Algorithmus funktioniert unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Worst-Case Eingabe.
  - Die Eingabe kann $\log n$ ($\log_2 n$)mal in zwei aufgeteilt werden kann. Zusätzlich benötigt der Algorithmus zum Kombinieren von den Teillisten n
  - Es ergibt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

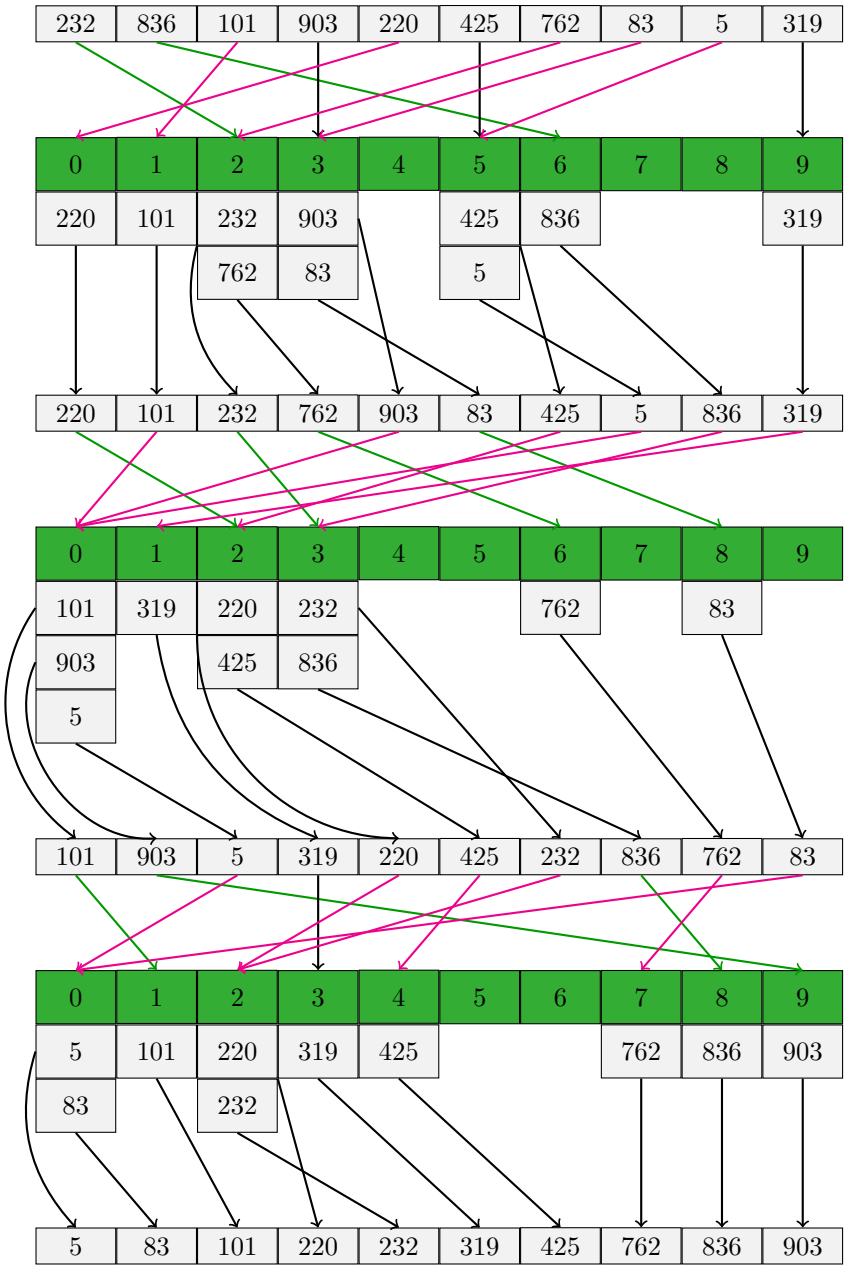
4.6 Radix Sort

```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:

- `newn`: 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() {
        return arr[top--];
        // Removes and returns the top element
    }

    void push(int k) {
        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.

5.2 Queues

Queues werden normalerweise mit den folgenden Funktionen erstellt:

- `newn`: 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) {
        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() {
        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;
        }
    }
}
```

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. Eine einfache Linked List kann wie folgt implementiert werden:

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

    public LinkedElement(Integer key) {
        this.key = key;
    }
}
```

```
class LinkedList {
    LinkedElement head = null;

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

    void delete(int k) {
        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;
        }
        else {
            head = curr.next;
        }
    }

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

5.4 Binary Search Tree

```
class BSTNode {
    Integer key;
    BSTNode left;
    BSTNode right;
    BSTNode parent;

    BSTNode(Integer k) {
        key = k;
    }
}
```

```
class BSTree {

    BSTNode root = null;

    void insert(BSTNode z) {
        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
    }
```

```
    void delete(BSTNode z) {
        if (z.left == null) { // If z has no left child, transplants the right child to z's position
            transplant(z, z.right);
        } else if (z.right == null) { // If z has no right child, transplants the left child 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) {
                y = y.left;
            } // Finds the next biggest element of z = smallest in right subtree of z
            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) {
        // 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;
        }
    }
```

```
    BSTNode iterativeSearch(int k) {
        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) {
        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) {
        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
    }
}
```

5.5 Red-Black Tree

```
class RBNode {
    Integer key;
    RBNode left;
    RBNode right;
    RBNode parent;
    Color color;

    RBNode(Integer k) {
        key = k;
    }
}
```

```
class RBTree {
    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

    void insert(RBNode z) {
        // 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
    }
```

```
    void fixColorsAfterInsertion(RBNode z) {
        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) { // If sibling exists and is red
                    z.parent.color = Color.BLACK; // Set z's parent to black
                    y.color = Color.BLACK; // Set z's sibling 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 { // If z doesn't have a sibling or 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 {
                // Same as above but with right and left exchanged
                RBNode y = z.parent.parent.left;
                if (y != null && y.color == Color.RED) {
                    z.parent.color = Color.BLACK;
                    y.color = Color.BLACK;
                    z.parent.parent.color = Color.RED;
                    z = z.parent.parent;
                } else {
                    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
        // Never needs to check nodes below z as their properties will not change after insertion
    }
```

```
    void rotateLeft(RBNode x) {
        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) {
        // 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;
    }
```

```
    void delete(RBNode z) {
        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) { // If 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 z to null
        } else if (z.left == null) { // If z only has a right child
            RBNode y = z.right;
            transplant(z, z.right);
            y.color = z.color;
        } else if (z.right == null) { // If z only has a left child
            RBNode y = z.left;
            transplant(z, z.left);
            y.color = z.color;
        } else { // If z has two children
            RBNode y = z.right;
            a = y;
            boolean wentLeft = false;
            while (y.left != null) { //
                a = y;
                y = y.left;
                wentLeft = true;
            }
            if (y.parent != z) { // Loop didn't run
                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);
        }
    }
```

```
    void fixColorsAfterDeletion(RBNode a, int dbh) {
        if (dbh == -1) { // Extra black node on the right
            RBNode x = a.left;
            RBNode b = a.right;
            RBNode c = b.left;
            RBNode d = b.right;
            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;
            RBNode d = b.left;
            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;
            }
        }
    }

    void transplant(RBNode u, RBNode v) {
        // Transplants v to the parent 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 Fortgeschrittene Datenstrukturen

7 Probabilistische Datenstrukturen

8 Graphen Algorithmen