Algorithmen und Datenstrukturen

Alles von Sortier- bis Suchalgorithmen und Graphentheorie.

Leon Muscat

Inhaltsverzeichnis

0 Einleitung

Im Kurs wird ein fundamentales Grundwissen über ein breites Spektrum von Datenstrukturen und Algorithmen, die in der Informatik eine hohe Relevanz haben, erarbeitet. Gelehrt werden die wichtigsten Sortier- (Quicksort, Mergesort, Heapsort, Radix Sort), Such- (Binäre Suchbäume, Rot-Schwarz-Bäume, Hashtabellen) und Graphenalgorithmen (Breitensuche, Tiefensuche, Prim, Kruskal Djikstra), mit Implementierungen in Java. Dazu wird das Abschätzen von Laufzeit, Komplexität und Speicherbedarf der Algorithmen behandelt.

0.1 Lernziele

- Die Studierenden haben Grundkenntnisse über Datenstrukturen
- Die Studierenden haben Grundkenntnisse über Algorithmen zum Suchen und Sortieren
- Die Studierenden haben Grundkenntnisse über Algorithmen zur Wegsuche in gerichteten und ungerichteten Graphen
- Die Studierenden haben die Fähigkeit die Laufzeit und Komplexität von Algorithmen zum Suchen und Sortieren abzuschätzen
- Die Studierenden haben die Fähigkeit die Laufzeit und Komplexität von Algorithmen zur Wegsuche in gerichteten und ungerichteten Graphen abzuschätzen
- Die Studierende lernen die Bedeutung dieser Algorithmen innerhalb der Informatik kennen mit besonderem Bezug auf Anwendungen im Bereich der “unternehmerischen Informatik”. Sie können abschätzen für welche Geschäftsprobleme die Algorithmen zur Anwendung kommen können.

0.2 Prüfung

40% der Note ergibt sich aus den Übungen.

60% der Note ergibt sich aus der zentralen Prüfung.

0.3 Literatur

“Algorithm” by Sedgewick, Roberst (Pearson Education, 2001)

Auch online einlesbar!

1 Dynamische Konnektivität

Gegeben ist eine Menge von N Objekten bzw. Nodes, die durch `UF(int N)` initialisiert wird. Der `Union(int a, int b)`-Befehl verbindet 2 Objekte und der `Connected(int a, int b)`-Befehl gibt zurück, ob es einen Weg gibt, der 2 Objekte verbindet.

Wir nehmen an, dass es sich bei einer Verbindung, um eine Äquivalenzrelation handelt. Das heißt, die Verbindung ist…

- reflexiv (p ist mit p verbunden),
- symmetrisch (wenn p mit q verbunden ist, dann ist q mit p verbunden) und
- transitiv (Wenn p mit q verbunden ist und q mit r verbunden ist, dann ist p mit r verbunden)

Die Verbindung von Nodes kann sich jederzeit verändern. Die Konnektivität ist also dynamisch.

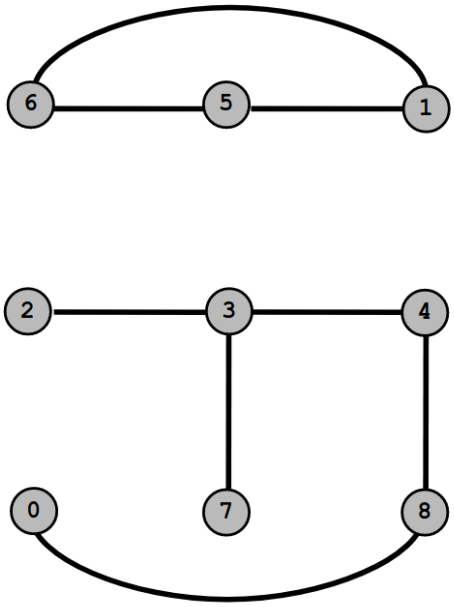


Abbildung 1: Dynamische Konnektivität

1.1 Quick-Find

Quick-Find ist ein aktiver Ansatz (eager approach). Die Idee ist ein Array zu bilden, in dem steht, ob ein Objekt mit einem anderen Objekt verbunden ist.

1.1.1 Datenstruktur

Integer Array `id[]` mit Größe `N`. An dem Index `i` steht, mit welchem Objekt, das Objekt `i` verbunden ist.

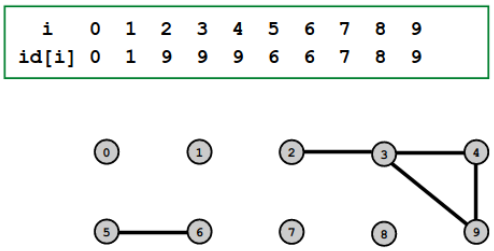


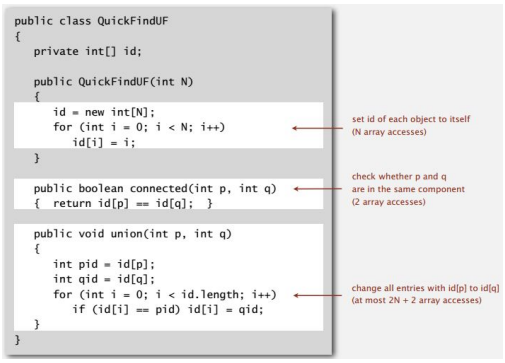
Abbildung 2: Quick-Find Array `id[]`

Ein Objekt kann auch mit sich selber verbunden sein. Dann gilt `i = id[i]`.

Um festzustellen, ob p und q miteinander verbunden sind, überprüft man, ob id[p] = id[q].

Zum Zusammenführen von Elementen, die p und q enthalten, müssen alle Elemente, deren `id` gleich `id[p]` ist, in `id[q]` ändern. Das kann bei vielen Elementen sehr aufwendig sein. Und das ist, wie wir sehen werden, ein kleines Problem, wenn wir eine große Anzahl von Objekten haben, weil es eine Menge Werte gibt, die sich ändern können. Aber trotzdem ist es einfach zu implementieren.

1.1.2 Java Implementierung



1.1.3 Vorteile

- Sehr schnelle Überprüfung, ob 2 Objekte verbunden sind (O(1)).

1.1.4 Nachteile

- Union ist sehr langsam, da alle Objekte, die zu p zeigen, nun zu q zeigen müssen. Die Union von N Objekten dauert daher O(N^2).

Insgesamt sind die Anzahl an Array Lese- und Schreibzugriffe also:

Algorithmus	Initialisierung	Union	Find
Quick Find	N	N	1

1.2 Quick Union

Quick Union ist eine Alternative zu Quick Find. Der Ansatz ist, die Elemente als Baum zu sortieren.

Aufgabe: Basierend auf id[] die Bäume Zeichnen (11_DynamischeKonnektivität.pdf S. 61)

1.2.1 Datenstruktur

Quick Union verwendet dieselbe Datenstruktur oder Array-ID mit der Größe N (siehe Abbildung 3), hat aber jetzt eine andere Interpretation. Wir stellen uns vor, dass dieses Array eine Reihe von Trees repräsentiert, die als Forest bezeichnet wird, wie in Abbildung 4 dargestellt. Jeder Eintrag im Array enthält also einen Verweis auf seine Eltern im Baum. Zum Beispiel ist der Elternteil von 3 vier, der Elternteil von 4 ist neun. Der Eintrag von 3 ist also vier und der Eintrag von 4 ist neun in dem Array. Jedem Eintrag in dem Array ist jetzt eine Wurzel zugeordnet, die gegeben ist durch `id[id[id[...]]]`. Das ist die Wurzel des Baums.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

Abbildung 3: Array ID für Quick Union

Um zu überprüfen, ob p und q verbunden sind, schaut man ob sie die selbe Wurzel haben. Siehe Abbildung 5.

Der `Union`-Befehl funktioniert, in dem die Wurzeln des Elements p and die Wurzel des Elements q gehangen wird. Siehe Abbildung 6.

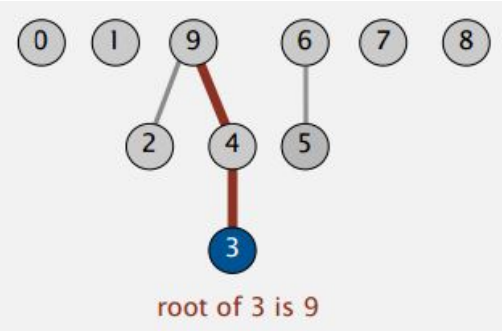


Abbildung 4: Darstellung in Form von Trees

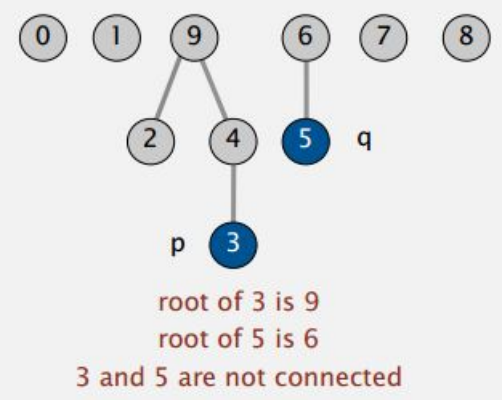


Abbildung 5: Überprüfung einer Verbindung mit Quick Union

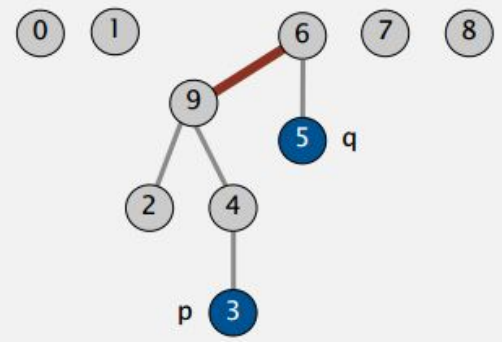


Abbildung 6: Quick Union `union(3, 5)`

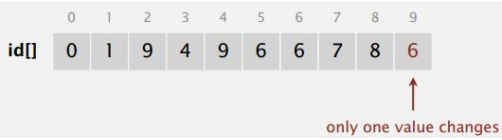
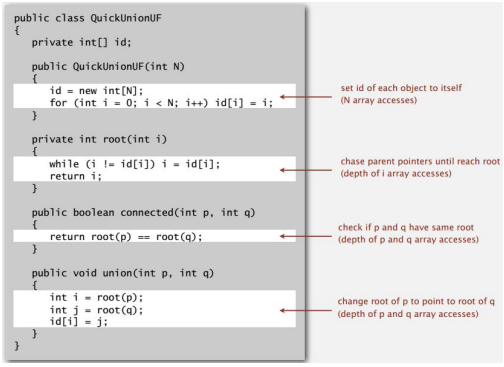


Abbildung 7: Quick Union Array for `union(3, 5)`

1.2.2 Java Implementierung



1.2.3 Vorteile

Der `Union`-Befehl ist sehr effektiv, da nur ein Wert im Array geändert werden muss.

1.2.4 Nachteile

Die Bäume können sehr groß werden, wodurch die Tiefe sehr groß wird. Dadurch dauert es (im worst case) lange bis die Wurzel gefunden wird.

Der `Find`-Befehl kann bis zu N Zugriffe auf das Array brauchen, da man den Baum entlang bis zur Wurzel gehen muss. Das ist besonders kostenintensiv, wenn der Baum sehr tief ist.

Insgesamt braucht Quick Union also mehr Array Lese- und Schreibzugriffe:

Algorithmus	Initialisierung	Union	Find
Quick Find	N	N	1
Quick Union	N	N (*)	N (worst case)

$\star$: Beinhaltet die Kosten für die Suche nach Wurzeln.

1.3 Verbesserung von Quick Union

Quick Find und Quick Union wurden eingeführt. Beide sind einfach zu implementieren. Aber sie können einfach keine großen dynamischen Verbindungsprobleme unterstützen. Wie können wir es also besser machen?

1.3.1 Gewichtetes Quick Union

Wir modifizieren Quick Union, s.d. keine großen Bäume entstehen können. Das ist umsetzbar, in dem immer der kleinere Baum an den größeren Baum gehangen wird. Siehe Abbildung 8. Dafür schreiben wir uns in einem 2. Array die Größe der Bäume mit.

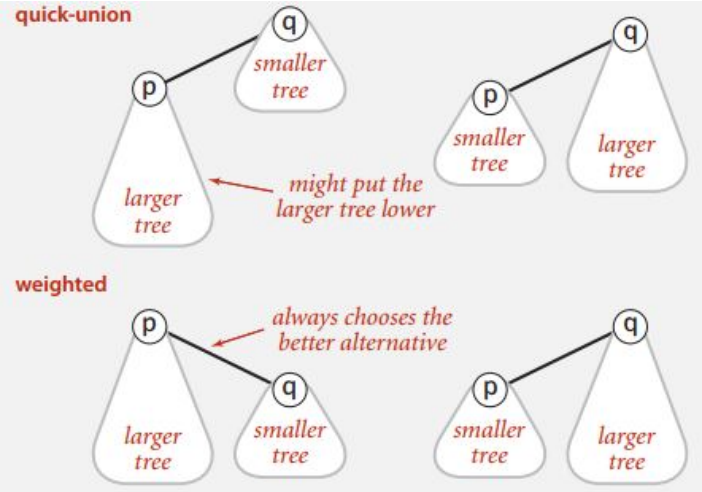


Abbildung 8: Gewichtetes Quick Union

Durch diese Änderung bleiben die Bäume flacher, wodurch der Weg zur Wurzel kürzer ist.

Theorem: Der Abstand eines Elements bis zur Wurzel ist maximal $\log N$.

Algorithmus	Initialisierung	Union	Verbunden
Quick Find	N	N	1
Quick Union	N	N *	N
Weighted QU	N	$\lg N$ *	$\lg N$

$*$: Beinhaltet die Kosten für die Suche nach Wurzeln.

1.3.1.1 Java

In der Java-Implementierung muss nur die `union`-Funktion geändert werden.

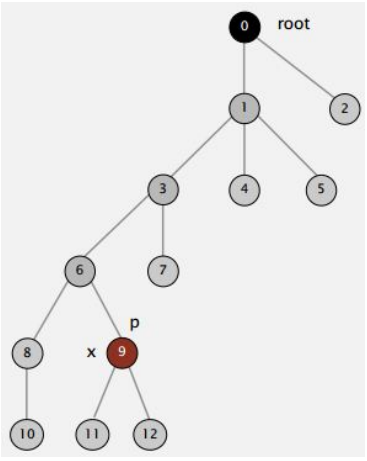
```java
public void union(int p, int q) {
    int i = root(p);
    int j = root(q);

    if (i == j) return;

```

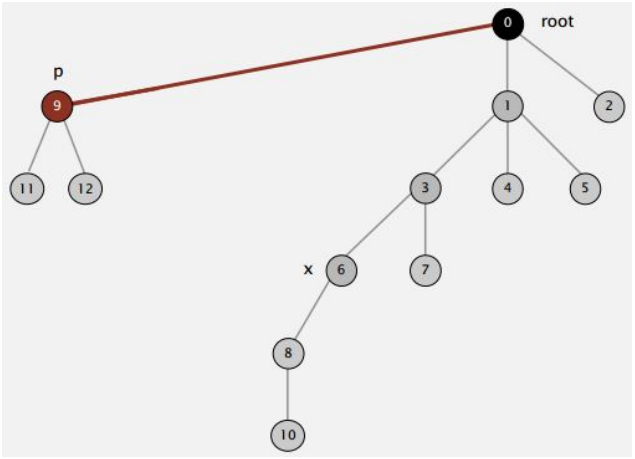
```
7       if (sz[i] < sz[j]) {
8           id[i] = j;
9           sz[j] += sz[i];
10      } else {
11          id[j] = i;
12          sz[i] += sz[j];
13      }
14  }
```

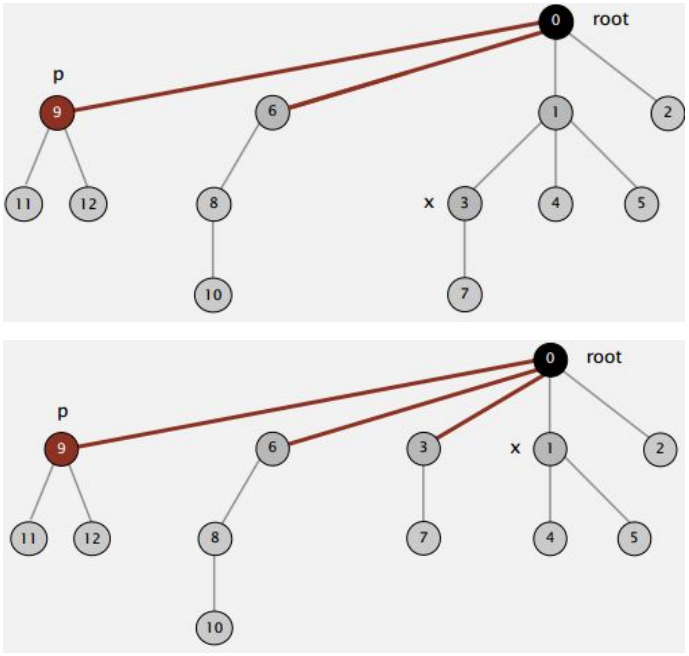
1.3.2 Pfad-Komprimierung

Gleich nach der Berechnung der Wurzel von p wird die `id` jedes untersuchten Knotens so gesetzt, dass sie auf diese Wurzel zeigt. Dadurch sind die Bäume sehr flach. Außerdem, sind die zusätzlichen Kosten der Pfad-Komprimierung konstant.

Beispiel: Nachdem wir die Wurzel von P gefunden haben, können wir genauso gut zurückgehen und jeden Knoten auf diesem Pfad auf die Wurzel zeigen lassen (wie in der Abbildung unten zeigt).







1.3.2.1 Java In der Java-Implementierung muss die `root`-Funktion modifiziert werden.

Two-pass Implementierung: Füge eine zweite Schleife hinzu, um die `id[]` von jedem besuchten Knoten auf die Wurzel zu setzen. Dadurch hat man aber eine zweite Schleife.

One-pass Variante: Lasse jeden anderen Knoten im Pfad auf seinen Großvater zeigen (und dadurch die Pfadlänge halbieren):

```
1  private int root(int i) {
2      while (i != id[i]) {
3          id[i] = id[id[i]];
4          i = id[i];
5      }
6
7      return i;
8  }
```

1.3.3 Weighted Quick Union with Path Compression (WQUPC)

Theorem: Eine Folge von M `Union`- und `Find`-Operationen benötigt $O(N + M \lg^* N)$ Zeit.

$\lg^*$ ist die iterierter Logarithmus. Der gibt an, wie oft die Logarithmusfunktion iteriert werden muss, bis die resultierende Zahl kleiner als 1 ist. Diese Funktion wächst sehr langsam.

N	lg*N
1	0
2	1
4	2
16	3
65536	4
2^65536	5

Die Funktion wächst so langsam, dass sie praktisch linear ist. Das bedeutet, dass WQUPC auch praktisch linear ist und ermöglicht es Probleme zu lösen, die sonst nicht behandelt werden können.

Die folgende Tabelle zeigt die Worst-case Laufzeit für M `Union-Find` Operationen auf einer Menge von N Elementen.

Algorithmus	Worst-case Zeit
quick-find	$M \cdot N$
quick-union	$M \cdot N$
weighted Quick-Union	$N + M \cdot \log N$
QU + path compression	$N + M \cdot \log N$
weighted QU + path compression	$N + M \cdot \lg^* N$

1.4 Anwendungen

Es gibt eine Vielzahl von Anwendungen, wie

- Perkolation
- Spiele (Go, Hex)
- Dynamische Konnektivität
- Least Common Ancestor Algorithmus
- …

1.4.1 Perkolation

Perkolation ist ein Modell für viele (physikalische) Systeme, wie eine Monte Carlo Simulation. Man erstellt ein Gitter mit $N \times N$ Positionen. Jede Position hat die Wahrscheinlichkeit p offen zu sein. Das System perkoliert, wenn es zwischen Oberseite und Unterseite eine Verbindung durch offene Positionen gibt. Siehe Abbildung 9.

Um zu Überprüfen, ob ein System perkoliert, kann das System in ein Dynamisches System übertragen werden. Aneinanderliegende offene Positionen bilden eine Verbindung. Das System perkoliert, wenn die oberste und unterste Reihe verbunden sind (Siehe Abbildung 10). Bei diesem Ansatz muss allerdings die Konnektivität mit jeder Position der oberen Reihe mit der unteren Reihe überprüft werden. Besser ist es, eine virtuelle obere und untere Position zu schaffen, die alle oberen bzw. untere Positionen verbindet. Dann muss nur überprüft werden, ob eine Verbindung von der virtuelle obere Position zu der virtuellen unteren Position existiert (Siehe Abbildung 11).

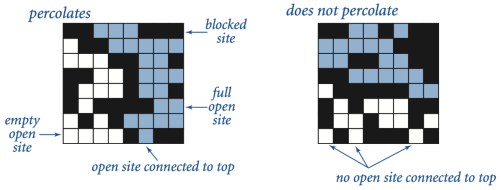


Abbildung 9: Perkolation mit $N = 8$

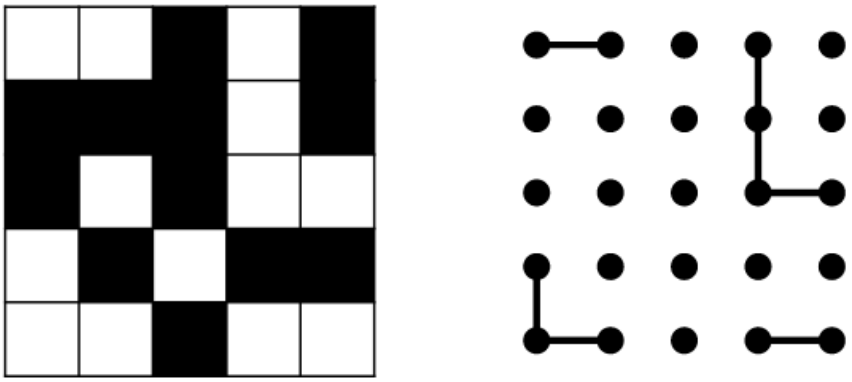


Abbildung 10: Prüfung auf Perkolation mit $N = 5$

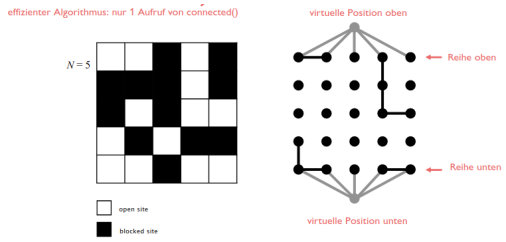
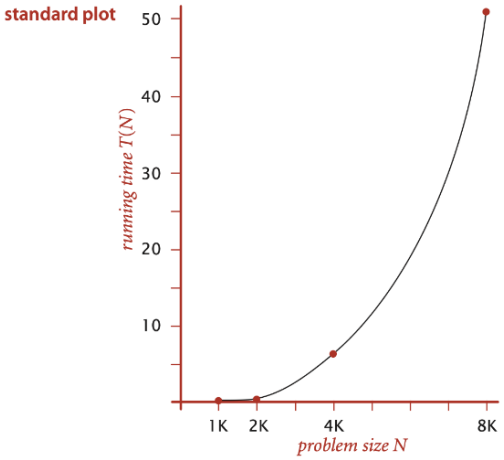


Abbildung 11: Prüfung auf Perkolation mit $N = 5$ mit virtuellen Positionen

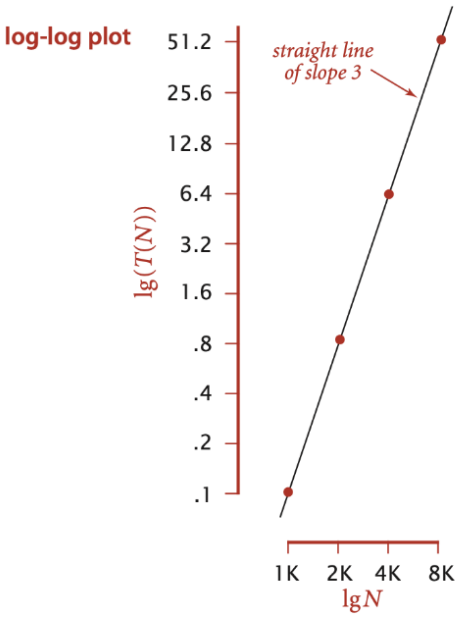
2 Analyse von Algorithmen

2.1 Beobachtung

Um Algorithmen durch Beobachtung zu analysieren, muss zunächst der Algorithmus implementiert werden. Dann erstellen wir Inputs verschiedener Größen und messen die Zeit, die der Algorithmus für das Verarbeiten benötigt. Aus den gemessenen Zeiten lässt sich ein Graph zeichnen:



Was wir wissen wollen, ist, was die Steigung des Graphen ist. Dafür erstellen wir einen Log-log Plot, indem wir den $\log_2$ von beiden Achsen nehmen (in der Abbildung steht $\lg$, aber gemeint ist $\log_2$):



Jeder Datenpunkt ist gegeben durch aN^b. Wir wollen also a und b berechnen. Da wir den $\log_2$ von jedem Datenpunkt wählen, ersetzen wir $a = 2^c$, da so die Rechnung leichter ist. Jetzt wählen wir 2 Datenpunkte…

1. $N_1 = 4000$ und $T(N_1) = 6.4s$
2. $N_2 = 8000$ und $T(N_2) = 51.1s$

… und setzen dieses in die folgende Gleichung ein:

$$\begin{cases} \log_2(T(N_1)) = b \cdot \log_2(N_1) + c \\ \log_2(T(N_2)) = b \cdot \log_2(N_2) + c \end{cases} \tag{1}$$

Dadurch erhalten wir $b = 2.997$ und $c = -33.1855$. Dadurch lässt sich die Hypothese erstellen, dass die Laufzeit ungefähr $1.024 \cdot 10^{-10} \cdot N^{2.997}$ Sekunden ist.

Eine Annäherung durch eine solche Berechnung ist sehr leicht zu erstellen, allerdings ist sie ungenau. Daher gibt es mathematische Modelle zur Berechnung der Laufzeit.

2.2 Mathematische Modelle

Gegeben ist ein Algorithmus, wie

```
1  int count = 0
2  for (int i = 0; i < N; i++)
3      for (int j = i+1; j < N; j++)
```

```
4           if (a[i] + a[j] == 0)
5               count++;
```

Als Erstes müssen wir alle Operationen in Abhängigkeit von N notieren. Daraus entsteht folgende Tabelle:

Operationen	Häufigkeit
Variablendeklaration	$N + 2$
Zuweisung	$N + 2$
Kleiner als Vergleich	$\frac{1}{2}(N + 1)(N + 2)$
Ist gleich Vergleich	$\frac{1}{2}N(N - 1)$
Arrayzugriff	$N(N - 1)$
Inkrement	$\frac{1}{2}N(N - 1)$ bis $N(N - 1)$

Jede Operationen x dauert eine bestimmte Anzahl an Nanosekunden c_x. Diese Zeit hängt von dem Compiler und Rechner ab. Daraus ergibt sich folgende Gleichung für die Laufzeit:

$$T(N) = c_1\text{Variablendeklarationen} + c_2\text{Zuweisungen} + \dots$$

Es ist aufwendig alle Operationen zu berücksichtigen, daher nehmen wir Array Zugriffe als Kostenmodell. Das bedeutet, wir berechnen die Laufzeit nur mit Hilfe der Array Zugriffe.

Bei großen Werten für N spielt nur die höchste Potenz eine Rolle, daher streichen wir alle Terme niedriger Ordnung weg. Diese Notation wird Tilde Notation genannt. Die Tabelle sieht wie folgt aus:

Operationen	Häufigkeit	Tilde Notation
Variablendeklaration	$N + 2$	N
Zuweisung	$N + 2$	N
Kleiner als Vergleich	$\frac{1}{2}(N + 1)(N + 2)$	$\frac{1}{2}N^2$
Ist gleich Vergleich	$\frac{1}{2}N(N - 1)$	$\frac{1}{2}N^2$
Arrayzugriff	$N(N - 1)$	N^2
Inkrement	$\frac{1}{2}N(N - 1)$ bis $N(N - 1)$	$\frac{1}{2}N^2$ bis N^2

Durch das Kostenmodell und die Tilde Notation kommen wir auf die Gleichung

$$T(N) = cN^2.$$

Zusammenfassend sind mathematische Modelle zur Berechnung der Laufzeit zwar akkurat, aber gleichzeitig komplex. Mithilfe von Annäherungen lässt sich die Berechnung allerdings vereinfachen.

2.3 Order-of-Growth Klassifikation

Wurde schon in GDI behandelt, daher nur eine Kurzfassung.

Die Laufzeit von jedem Algorithmus wird nach dem höchsten Exponenten klassifiziert (dabei werden Koeffizienten vernachlässigt). Es gibt folgende Klassen:

$$1, \log N, N, N \log N, N^2, N^3, 2^N$$

(Ich weiß nicht, warum N^3 dazugekommen ist.)

Prinzipiell ist die Tilde Notation zu bevorzugen, da die Big-O-Notation nur eine Obergrenze bietet.

2.4 Abhängigkeit vom Input

Es gibt verschiedene Typen von Laufzeitanalyse, nämlich…

- **Best case** bzw. Untergrenze der Kosten. Der Best case ergibt sich durch den einfachsten Input.
- **Worst case** bzw. Obergrenze der Kosten. Der Worst case ergibt sich durch den schwierigsten Input.
- **Average case** bzw. durchschnittliche erwarteten Kosten. Der Average case gibt die Kosten für einen zufälligen Input an und ermöglicht es Performance vorherzusagen.

2.5 Speicher

Es ist auch möglich, den Speicher als Kostenmodell zu nutzen.

In Java sind die Speicherkosten für primitive Typen:

Typ	Byte
byte	1

Typ	Byte
char	2
int	4
float	4
long	8
double	8

für eindimensionale Arrays:

Typ	Byte
char[]	2N + 24
int[]	4N + 24
double[]	8N + 24

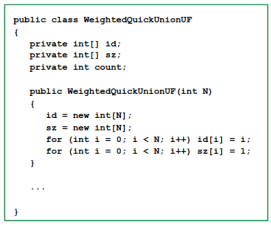
und für zweidimensionale Arrays:

Typ	Byte
char[][]	~ 2 M N
int[][]	~ 4 M N
double[][]	~ 8 M N

Außerdem hat jedes Objekt einen Overhead von 16 Bytes, Referenzen von 8 Bytes und Padding, um auf ein Vielfaches von 8 bytes zu kommen.

Es gibt 2 Methoden der Analyse des Speicherverbrauchs: Shallow memory usage und Deep memory usage. Bei Shallow memory usage zählen referenzierte Objekte nicht zur Speichernutzung hinzu, aber bei Deep memory usage schon. In beiden Methoden werden statische Variablen (die Variablen, die nur einmal pro Klasse existieren) nicht beachtet.

Beispiel:



```
public class WeightedQuickUnionUF
{
   private int[] id;
   private int[] sz;
   private int count;

   public WeightedQuickUnionUF(int N)
   {
      id = new int[N];
      sz = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      for (int i = 0; i < N; i++) sz[i] = 1;
   }

   ...

}
```

16 byte (object overhead)

8 byte (Referenz auf array)
8 byte (Referenz auf array)
4 byte (int)

4 byte (padding)

4N + 24 byte (int[] array)
4N + 24 byte (int[] array)

8N + 88 byte

3 Datenstrukturen

Eine Datenstruktur ist eine logische Anordnung von Daten mit Zugriffs- und Verwaltungsmöglichkeiten der repräsentierten Informationen über Operationen. Eine Datenstruktur besitzt:

- Implementierung: Code, der die Operationen implementiert
- Interface: Beschreibung von Datentypen und Basisoperationen
- Client: Programm, das Operationen verwendet, die im Interface definiert sind

3.1 Stack

Basiert auf dem LIFO (Last In First Out) Prinzip.

Jeder Stack hat einen Stack-Pointer, der auf das höchste Element zeigt (wird aber nicht in der Vorlesung behandelt).

Das Interface ist simpel:

`push()`: Legt ein neues Element oben auf den Stack `pop()`: Nimmt das oberste Element vom Stack `size()`: Anzahl an Elementen im Stack

Normalerweise wird durch `pop` das oberste Element nicht nur ausgegeben, sondern auch entfernt. Ist dies nicht der Fall, dann spricht man von Loitering.

3.1.1 Implementierung

Wir schauen uns 3 Implementierungen an: Verkettete Listen und Arrays und dynamische Arrays.

3.1.1.1 Verkettete Listen Bei verketteten Listen ist das letzte Element (also das, welches ein null Pointer hat), das höchste Element des Stacks. Der Anfang der verketteten Liste ist also der Rumpf des Stacks. Jede Operation braucht im Worst-Case eine konstante Zeit und verbraucht für N Knoten (mit String Datentyp) $40N$ Bytes (16 Objekt Overhead, 8 Bytes innere Klasse Overhead, 8 Bytes Referenz auf String, 8 Bytes Referenz auf Knoten).

3.1.1.2 Arrays Bei einem Array `s[N]` kann es maximal N Elemente im Stack geben. Ist der Stack leer und man versucht ein weiteres Element zu nehmen, dann wird ein Underflow Fehler geworfen. Ist der Stack voll und man versucht ein weiteres Element hinzuzufügen, dann entsteht ein Overflow Fehler. Ein Array als Stack ermöglicht das Einfügen von null Elementen

3.1.1.3 Dynamische Arrays Dynamische Arrays funktionieren, in dem das Array kopiert wird, sobald `push` oder `pop` aufgerufen wird. Das ist aber sehr teuer und führt zu N^2 Arrayzugriffen für das Einfügen der ersten N Elemente.

Es gibt aber auch eine effiziente Lösung. Die Größe des Arrays wird verdoppelt, wenn das Array voll ist und das Array wird halbiert, wenn das Array ein Viertel voll ist. Das Array ist also immer zwischen 25% und 100% voll. Die Laufzeit ist also im Best-Case für `push` und `pop` 1, im Worst-Case N und amortisiert (= Durchschnittliche Laufzeit pro Operation für eine Worst-Case Sequenz an Operationen) auch 1. Dabei wird zwischen $8N$ und $32N$ Speicher verbraucht.

3.2 Queue

Eine Warteschlange, die nach dem FIFO (First In First Out) Prinzip funktioniert.

Das Interface ist wieder simpel:

- `enqueue()`: Fügt ein neues Element am Ende der Queue hinzu
- `dequeue()`: Nimmt das erste Element aus der Queue
- `size()`: Gibt die Länge der Queue zurück

3.2.1 Implementierung

Ähnlich wie bei dem Stack kann eine Queue durch verkettete Listen, Arrays und dynamische Arrays implementiert werden.

3.2.1.1 Verkettete Listen Hierbei ist wichtig zu beachten, dass das erste Element der Liste, das vorderste Element der Queue ist. Durch `dequeue` wird also das letzte Element ausgegeben und gelöscht (dadurch geht der Pointer zum nächsten Element verloren → nächstes Element wird erstes Element). Bei `enqueue` wird ein neues Element am Ende der Liste hinzugefügt.

3.3 Priority Queues

Priority Queues sind abstrakte Datenstrukturen, die Elemente basierend auf ihrer Priorität speichern und verwalten. Sie unterstützen das Einfügen von Elementen und das Entfernen des Elements mit

der höchsten (oder niedrigsten) Priorität. Eine häufige Implementierung von Priority Queues ist die Max-PQ, die das Element mit der höchsten Priorität verwaltet.

3.3.1 API

```
public class MaxPQ<Key extends Comparable<Key>>
{
    public MaxPQ(); // Konstruktor
    public void insert(Key v); // Element einfügen
    public Key delMax(); // Element mit der höchsten Priorität
       entfernen und zurückgeben
    public boolean isEmpty(); // Prüft, ob die PQ leer ist
}
```

3.3.2 Implementierungen

Es gibt verschiedene Implementierungen von Priority Queues:

1. **Unsortierte Arrays oder Listen:** In dieser Implementierung wird das Element mit der höchsten Priorität durch Scannen der gesamten Datenstruktur gefunden. Das Einfügen hat eine Laufzeit von $O(1)$, während das Entfernen des maximalen Elements eine Laufzeit von $O(n)$ hat.
2. **Sortierte Arrays oder Listen:** Hier werden die Elemente in sortierter Reihenfolge gehalten. Das Einfügen hat eine Laufzeit von $O(n)$, da im schlimmsten Fall alle Elemente verschoben werden müssen, um Platz für das neue Element zu schaffen. Das Entfernen des maximalen Elements hat eine Laufzeit von $O(1)$.
3. **Binäre Heaps:** Eine effizientere Implementierung sind binäre Heaps, insbesondere binäre Max-Heaps. Sie bieten eine Laufzeit von $O(\log n)$ für das Einfügen und Entfernen von Elementen. Binäre Heaps sind vollständige binäre Bäume, die die Heap-Eigenschaft erfüllen (jeder Knoten hat einen höheren oder gleichen Wert als seine Kinder).

3.3.3 Andwendungen

Priority Queues sind in vielen Anwendungen und Algorithmen nützlich, wie z.B. bei der Verwaltung von Ereignissen in einer Simulation, bei der Suche nach den k nächstgelegenen Punkten, bei der Implementierung von Dijkstra's Algorithmus oder bei der Implementierung des Huffman-Codierungsalgorithmus.

3.3.4 Eigenschaften

- **Dynamische Größe:** Einige Implementierungen, wie z.B. Heaps, benötigen eine dynamische Größe, um effizient zu sein. In solchen Fällen kann die Größe des zugrunde liegenden Arrays bei Bedarf vergrößert oder verkleinert werden.
- **Stabilität:** Im Allgemeinen sind Priority Queues nicht stabil. Das bedeutet, dass die Reihenfolge von Elementen mit gleicher Priorität nicht gleich bleibt.

3.4 Generics

Java Generics ist eine leistungsstarke Funktion, die in Java 5 eingeführt wurde und die Verwendung von *Typ-Parametern* in Klassen, Schnittstellen und Methoden ermöglicht, wodurch Typsicherheit und Wiederverwendbarkeit von Code gewährleistet werden.

3.4.1 Konzepte

- **Typ-Parameter**: `<T>` ist ein Platzhalter für den tatsächlichen Typ, der bei der Erstellung einer Instanz angegeben wird.
- **Generische Klassen**: Klassen, die einen oder mehrere Typparameter akzeptieren, z.B. `class MyClass<T> { }`.
- **Generische Schnittstellen**: Schnittstellen, die einen oder mehrere Typparameter akzeptieren, z. B. `Interface MyInterface<T> { }`.
- **Generische Methoden**: Methoden, die einen oder mehrere Typparameter akzeptieren, z.B. `public <T> void myMethod(T t){ }`.

3.4.2 Vorteile

1. **Typensicherheit**: Generische Methoden erzwingen eine Typüberprüfung während der Kompilierung und verhindern typbezogene Laufzeitfehler.
2. **Wiederverwendbarkeit von Code**: Generische Klassen und Methoden können für verschiedene Typen wiederverwendet werden, wodurch Code-Duplizierung vermieden wird.
3. **Lesbarer Code**: Generische Klassen verbessern die Lesbarkeit von Code, indem sie Typinformationen explizit machen.

3.4.3 Beispiel

```java
// Generic class
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}

// Usage
Box<String> stringBox = new Box<>();
stringBox.setContent("Hello, Generics!");
String content = stringBox.getContent();
```

3.5 Iterables

Java Iterables sind ein Kernbestandteil des Java Collections Framework und bieten eine Schnittstelle zum sequenziellen Durchlaufen von Sammlungselementen. `Iterable` ist eine Schnittstelle, die von verschiedenen Sammlungsklassen wie `ArrayList`, `HashSet` und `LinkedList` implementiert wird.

3.5.1 Schlüsselkonzepte

- **Iterable Interface**: Eine Basisschnittstelle mit einer einzigen abstrakten Methode, `iterator()`, die ein `Iterator<T>`-Objekt zurückgibt.
- **Iterator Interface**: Stellt Methoden zum Durchlaufen einer Sammlung bereit, wie `hasNext()` und `next()`.

3.5.2 Anwendung bei Arrays

Obwohl Arrays das `Iterable`-Interface nicht direkt implementieren, können sie durch Verwendung von `Arrays.asList()` einfach in `List`-Objekte konvertiert werden, die iterierbar sind.

3.5.3 Anwendung bei Bags

Bags sind wie Arrays, allerdings ist die Reihenfolge der Elemente egal. Das Iterable Interface bietet genau die Funktionen, die für Bags gebraucht werden.

3.5.4 Beispiel

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class IterablesBeispiel {
    public static void main(String[] args) {
        // Erstelle ein Array und konvertiere es in eine Iterable-Liste
        Integer[] zahlenArray = {1, 2, 3, 4, 5};
        List<Integer> zahlenListe = Arrays.asList(zahlenArray);

        // Iteriere über die Liste mit einem Iterator
        Iterator<Integer> iterator = zahlenListe.iterator();
        while (iterator.hasNext()) {
            Integer zahl = iterator.next();
            System.out.println(zahl);
        }

        // Iteriere über die Liste mit einer for-each Schleife (
            Syntaktischer Zucker für Iterator)
        for (Integer zahl : zahlenListe) {
            System.out.println(zahl);
        }
    }
}
```

In diesem Beispiel wird ein Array von Ganzzahlen in ein `List`-Objekt konvertiert, das das `Iterable` -Interface implementiert. Die Elemente der Liste werden dann mit einem `Iterator` und einer for-each-Schleife durchlaufen.

3.6 Bäume

Bäume sind eine wichtige Datenstruktur in der Informatik, die hierarchische Beziehungen zwischen Elementen darstellen. Sie bestehen aus Knoten, die über Kanten miteinander verbunden sind. In der Regel hat jeder Knoten einen Elternknoten und mehrere Kindknoten. Die Knoten ohne Kindknoten werden als Blätter bezeichnet, während der Knoten ohne Elternknoten die Wurzel des Baums ist. Es gibt verschiedene Typen von Bäumen, die für unterschiedliche Anwendungsfälle optimiert sind, wie z.B. binäre Bäume und 2-3 Bäume.

3.6.1 Binäre Bäume

Ein binärer Baum ist ein Baum, bei dem jeder Knoten höchstens zwei Kindknoten haben kann: einen linken und einen rechten Kindknoten.

3.6.2 Selbstbalancierende binäre Suchbäume

Selbstbalancierende binäre Suchbäume sind binäre Suchbäume, die bei Einfügen oder Löschen von Elementen automatisch ihre Struktur anpassen, um die Höhe des Baums minimal zu halten. Zu den bekanntesten selbstbalancierenden binären Suchbäumen gehören AVL-Bäume und Rot-Schwarz-Bäume. Diese Bäume garantieren eine Laufzeit von $O(\log n)$ für Suche, Einfügen und Löschen.

3.7 Heaps

Heaps sind eine Art von abstrakten Datenstrukturen, die häufig zur Implementierung von Priority Queues verwendet werden. Sie sind binäre Bäume, die die Heap-Eigenschaft erfüllen, bei der die Priorität eines Knotens größer (oder kleiner) ist als die Priorität seiner Kinder. Binäre Heaps können als Max-Heaps (Elemente mit der höchsten Priorität an der Wurzel) oder Min-Heaps (Elemente mit der niedrigsten Priorität an der Wurzel) implementiert werden.

3.7.1 Binäre Heaps

Binäre Heaps sind eine Art von Heap, bei denen jeder Knoten höchstens zwei Kinder hat. Sie sind vollständige binäre Bäume, was bedeutet, dass alle Ebenen bis auf die letzte vollständig gefüllt sind und die letzte Ebene von links nach rechts gefüllt ist. Binäre Heaps können effizient in Arrays gespeichert werden, wobei der Elternknoten eines Knotens an Index i bei Index $\lfloor \frac{i}{2} \rfloor$ liegt und die beiden Kinder bei den Indizes $2i$ und $2i + 1$.

3.7.2 Einfügen und Entfernen von Elementen

- **Einfügen:** Um ein Element in einen Heap einzufügen, wird es zunächst an die nächste freie Position am Ende des Heaps (Array) hinzugefügt. Anschließend wird das Element nach oben verschoben (swim), indem es wiederholt mit seinem Elternknoten vertauscht wird, bis die Heap-Eigenschaft erfüllt ist.
- **Entfernen:** Um das Element mit der höchsten (oder niedrigsten) Priorität zu entfernen, wird die Wurzel des Heaps entfernt und das letzte Element des Heaps an die Wurzelposition verschoben.

Dann wird das Element nach unten verschoben (sink), indem es wiederholt mit dem Kind mit der höheren Priorität vertauscht wird, bis die Heap-Eigenschaft erfüllt ist.

3.7.3 Order-of-Growth Laufzeit

Die wichtigsten Operationen in Heaps, Einfügen und Entfernen, haben eine Laufzeit von $O(\log n)$, wobei n die Anzahl der Elemente im Heap ist. Dies liegt daran, dass die Höhe des binären Heaps $\log n$ beträgt und die Operationen swim und sink maximal die Höhe des Baumes betragen.

3.7.4 Immutability von Keys

In Heaps sollten die Schlüssel (Keys) unveränderlich sein, d.h. ihr Wert sollte sich nach dem Einfügen in den Heap nicht ändern. Andernfalls kann die Heap-Eigenschaft verletzt werden, was zu inkorrektem Verhalten führen kann. Wenn ein Key geändert werden muss, sollte das Element entfernt, der Key aktualisiert und das Element erneut eingefügt werden.

3.7.5 Underflow und Overflow

- **Underflow:** Ein Heap-Underflow tritt auf, wenn versucht wird, ein Element aus einem leeren Heap zu entfernen. Dies sollte in der Implementierung entsprechend behandelt werden, z.B. durch das Werfen einer Ausnahme oder das Zurückgeben eines Nullwertes.
- **Overflow:** Ein Heap-Overflow tritt auf, wenn der zugrunde liegende Array oder die Datenstruktur keine weiteren Elemente aufnehmen kann. In diesem Fall sollte die Größe des Arrays dynamisch angepasst werden, indem es vergrößert wird, um zusätzlichen Platz für neue Elemente bereitzustellen. Bei Verwendung eines dynamischen Arrays, wie zum Beispiel einer ArrayList in Java, wird dies automatisch gehandhabt.

Insgesamt sind Heaps eine wichtige Datenstruktur, die in vielen Anwendungen und Algorithmen verwendet wird, um Elemente basierend auf ihren Prioritäten effizient zu verwalten. Binäre Heaps bieten eine gute Balance zwischen Einfachheit, Speicherplatz- und Laufzeiteffizienz und sind die am häufigsten verwendete Heap-Implementierung.

4 Sortieralgorithmen

4.1 Insertion Sort

Insertion Sort ist ein einfacher Sortieralgorithmus, der auf der Idee basiert, die Elemente der Liste nacheinander an der richtigen Position in einer bereits sortierten Teilsequenz einzufügen. Der Algorithmus ist für kleine Datensätze oder für teilweise sortierte Datensätze geeignet.

4.1.1 Algorithmus

1. Beginne beim ersten Element der Liste.
2. Vergleiche das aktuelle Element mit den bereits sortierten Elementen in der Liste.
3. Finde die richtige Position für das aktuelle Element in der sortierten Teilsequenz und füge es dort ein.
4. Fortfahren mit dem nächsten Element und wiederholen, bis alle Elemente sortiert sind.

4.1.2 Java Implementierung

```java
public class Insertion {
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
            for (int j = i; j > 0; j--)
                if (less(a[j], a[j-1]))
                    exch(a, j, j-1);
                else break;
    }
}
```

4.1.3 Laufzeitanalyse

- **Best-Case:** Der Best-Case tritt auf, wenn die Eingabedaten bereits vollständig sortiert sind. In diesem Fall muss der Algorithmus nur einmal durch die Liste gehen und jedes Element mit seinem Vorgänger vergleichen, ohne Elemente zu verschieben. Die Best-Case-Laufzeit ist daher linear und beträgt $O(n)$, wobei n die Anzahl der Elemente in der Liste ist.
- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. In diesem Fall muss der Algorithmus etwa die Hälfte der bereits sortierten Elemente durchlaufen, um die korrekte Position für das aktuelle

Element zu finden. Daher führt der Algorithmus etwa $n/2$ Vergleiche und Verschiebungen für jedes Element durch. Die Average-Case-Laufzeit beträgt demnach $O(n^2)$.

- **Worst-Case:** Der Worst-Case tritt auf, wenn die Eingabedaten in umgekehrter Reihenfolge sortiert sind. Der Algorithmus muss in diesem Fall jedes Element mit allen bereits sortierten Elementen vergleichen und verschieben, was zu einer quadratischen Anzahl von Vergleichen und Verschiebungen führt. Die Worst-Case-Laufzeit beträgt daher $O(n^2)$.

Da Insertion Sort bei großen Datensätzen oder bei ungeordneten Daten ineffizient ist, sollte man in solchen Fällen auf effizientere Sortieralgorithmen wie Merge Sort, Quick Sort oder Heap Sort zurückgreifen.

4.1.4 Eigenschaften

- **Stabilität:** Insertion Sort ist ein stabiler Sortieralgorithmus, da gleichwertige Elemente ihre relative Reihenfolge beibehalten.
- **In-Place:** Der Algorithmus ist in-place, da er keine zusätzlichen Datenstrukturen benötigt und die Sortierung innerhalb der gegebenen Liste durchführt.
- **Online:** Insertion Sort kann als Online-Algorithmus verwendet werden, da er in der Lage ist, neue Elemente während der Sortierung zu akzeptieren und zu verarbeiten.

4.2 Selection Sort

Selection Sort ist ein einfacher Sortieralgorithmus, der auf der Idee basiert, das kleinste (oder größte) Element der Liste zu finden und es an den richtigen Positionen zu platzieren. Der Algorithmus ist für kleine Datensätze geeignet, aber weniger effizient für größere oder teilweise sortierte Datensätze.

4.2.1 Algorithmus

1. Beginne beim ersten Element der Liste.
2. Suche das kleinste Element in der verbleibenden unsortierten Liste.
3. Tausche das kleinste Element mit dem Element an der aktuellen Position.
4. Fortfahren mit dem nächsten Element und wiederholen, bis alle Elemente sortiert sind.

4.2.2 Java Implementierung

```
1 public class Selection {
2     public static void sort(Comparable[] a)
```

```
3       {
4           int N = a.length;
5           for (int i = 0; i < N; i++)
6           {
7               int min = i;
8               for (int j = i+1; j < N; j++)
9                   if (less(a[j], a[min]))
10                      min = j;
11              exch(a, i, min);
12          }
13      }
14  }
```

4.2.3 Laufzeitanalyse

- **Best-Case:** Der Best-Case tritt auf, wenn die Eingabedaten bereits vollständig sortiert sind. In diesem Fall muss der Algorithmus dennoch die gesamte Liste durchlaufen und die kleinsten Elemente suchen, da er die Sortierung nicht erkennen kann. Die Best-Case-Laufzeit beträgt daher $O(n^2)$, wobei n die Anzahl der Elemente in der Liste ist.
- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. In diesem Fall muss der Algorithmus weiterhin die gesamte Liste durchlaufen und die kleinsten Elemente suchen. Die Average-Case-Laufzeit beträgt demnach $O(n^2)$.
- **Worst-Case:** Der Worst-Case tritt auf, wenn die Eingabedaten in umgekehrter Reihenfolge sortiert sind. Der Algorithmus muss in diesem Fall ebenfalls die gesamte Liste durchlaufen und die kleinsten Elemente suchen. Die Worst-Case-Laufzeit beträgt daher $O(n^2)$.

Da Selection Sort bei großen Datensätzen oder bei ungeordneten Daten ineffizient ist, sollte man in solchen Fällen auf effizientere Sortieralgorithmen wie Merge Sort, Quick Sort oder Heap Sort zurückgreifen.

4.2.4 Eigenschaften

- **Stabilität:** Selection Sort ist im Allgemeinen kein stabiler Sortieralgorithmus, da gleichwertige Elemente ihre relative Reihenfolge nicht unbedingt beibehalten. Eine stabile Variante des Selection Sort kann jedoch implementiert werden, indem anstelle des Tauschens die Elemente verschoben werden.
- **In-Place:** Der Algorithmus ist in-place, da er keine zusätzlichen Datenstrukturen benötigt und die Sortierung innerhalb der gegebenen Liste durchführt.

- **Online:** Selection Sort ist kein Online-Algorithmus, da er nicht in der Lage ist, neue Elemente während der Sortierung zu akzeptieren und zu verarbeiten.

4.3 Shell Sort

Shell Sort ist ein in-place Sortieralgorithmus, der auf dem Insertion Sort-Algorithmus basiert und dessen Effizienz durch die Einführung von sogenannten "Spaltungen" verbessert wird. Shell Sort ist besonders effektiv bei mittelgroßen Datensätzen und zeigt eine bessere Leistung als einfache Sortieralgorithmen wie Insertion Sort oder Selection Sort.

4.3.1 Algorithmus

1. Wähle eine Spaltungssequenz (z. B. die Sequenz von Sedgewick oder die Sequenz von Pratt).
2. Sortiere die Elemente innerhalb jeder Spalte mit Insertion Sort oder einer Variation davon.
3. Reduziere die Spaltungsgröße und wiederhole Schritt 2, bis die Spaltungsgröße auf 1 reduziert ist und die Liste vollständig sortiert ist.

4.3.2 Java Implementierung

```
1  public class Shell {
2      public static void sort(Comparable[] a)
3      {
4          int N = a.length;
5          int h = 1;
6          while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093,
               ...
7          while (h >= 1)
8          { // h-sort the array.
9              for (int i = h; i < N; i++)
10             {
11                 for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
12                     exch(a, j, j-h);
13             }
14
15             h = h/3;
16         }
17     }
18 }
```

4.3.3 Laufzeitanalyse

Die Laufzeit von Shell Sort hängt stark von der gewählten Spaltungssequenz ab. Im Allgemeinen ist die Laufzeit jedoch besser als die von einfachen Sortieralgorithmen wie Insertion Sort oder Selection Sort.

- **Best-Case:** Der Best-Case tritt auf, wenn die Eingabedaten bereits vollständig sortiert sind. Die Best-Case-Laufzet ist $O(n \log n)$.
- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. Die Average-Case-Laufzeit hängt von der Spaltungssequenz ab. Generell ist die Average-Case-Laufzeit $O(n \log n)$. Für eine Spaltungssequenz von $3x + 1$ ist die Average-Case-Laufzeit $O(n^{1.5})$. Insgesamt ist die Laufziet besser als die quadratische Laufzeit von Insertion Sort oder Selection Sort.
- **Worst-Case:** Der Worst-Case tritt auf, wenn die Eingabedaten in umgekehrter Reihenfolge sortiert sind. Die Worst-Case-Laufzeit kann im schlimmsten Fall $O(n^2)$ erreichen.

Shell Sort ist für mittelgroße Datensätze geeignet, bei großen Datensätzen oder bei ungeordneten Daten sollte man jedoch auf effizientere Sortieralgorithmen wie Merge Sort, Quick Sort oder Heap Sort zurückgreifen.

Besonders gut ist Shell Sort, wenn die Liste bereits teilweise sortiert ist und die falsch sortierten Elemente weit von ihrer eigentlich Position entfernt sind.

4.3.4 Eigenschaften

- **Stabilität:** Shell Sort ist im Allgemeinen kein stabiler Sortieralgorithmus, da gleichwertige Elemente ihre relative Reihenfolge nicht unbedingt beibehalten. Eine stabile Variante des Shell Sort kann jedoch implementiert werden, indem anstelle des Tauschens die Elemente verschoben werden und besondere Sorgfalt auf die Spaltungssequenz gelegt wird.
- **In-Place:** Der Algorithmus ist in-place, da er keine zusätzlichen Datenstrukturen benötigt und die Sortierung innerhalb der gegebenen Liste durchführt.
- **Adaptiv:** Shell Sort kann als adaptiver Algorithmus betrachtet werden, da seine Leistung von der anfänglichen Ordnung der Eingabedaten abhängt. Wenn die Eingabedaten teilweise sortiert sind, kann der Algorithmus schneller sortieren.
- **Spaltungssequenzen:** Die Wahl der Spaltungssequenz hat einen großen Einfluss auf die Leistung des Algorithmus. Einige bekannte Sequenzen sind die Sequenz von Shell (mit einer Spaltungsgröße von $n/2$, $n/4$, $n/8$ usw.), die Sequenz von Sedgewick (mit einer Spaltungsgröße von

$4^k + 3 * 2^{k-1} + 1$) und die Sequenz von Pratt (mit einer Spaltungsgröße von $2^i * 3^j$). Eine optimale Spaltungssequenz ist noch nicht bekannt (oder noch nicht bewiesen).

4.4 Merge Sort

Merge Sort ist ein leistungsfähiger und stabiler Sortieralgorithmus, der auf dem Prinzip des "Teilen und Herrschen" basiert. Er teilt die Eingabeliste wiederholt in zwei Hälften, sortiert diese rekursiv und führt sie dann wieder zusammen. Merge Sort eignet sich gut für große Datensätze und zeigt eine bessere Leistung als einfache Sortieralgorithmen wie Insertion Sort oder Selection Sort.

4.4.1 Algorithmus

1. Teile die Liste in zwei gleich große Hälften.
2. Sortiere beide Hälften rekursiv mit Merge Sort.
3. Führe die sortierten Hälften wieder zusammen, indem du die Elemente in der richtigen Reihenfolge auswählst.

Eine Variante von Merge Sort ist Bottom-up Merge Sort.

1. Beginne mit Teilsequenzen der Größe 1.
2. Führe jeweils zwei benachbarte Teilsequenzen der aktuellen Größe zusammen.
3. Verdopple die Größe der Teilsequenzen und wiederhole Schritt 2, bis die gesamte Liste sortiert ist.

Dadurch ist keine Rekursion nötig. An der Laufzeit ändert sich aber nichts.

4.4.2 Java Implementierung

```java
public class MergeSort {
    private static void sort(Comparable[] a, Comparable[] aux, int lo,
        int hi)
    {
        if (hi <= lo) return;

        int mid = lo + (hi - lo) / 2;
        sort (a, aux, lo, mid);
        sort (a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    {
```

```
14          aux = new Comparable[a.length];
15          sort(a, aux, 0, a.length - 1);
16      }
17
18      private static void merge(Comparable[] a, Comparable[] aux, int lo,
            int mid, int hi)
19      {
20          assert isSorted(a, lo, mid); // precondition: a[lo..mid] sorted
21          assert isSorted(a, mid+1, hi); // precondition: a[mid+1..hi]
               sorted
22
23          for (int k = lo; k <= hi; k++)
24              aux[k] = a[k];
25
26          int i = lo, j = mid+1;
27          for (int k = lo; k <= hi; k++)
28          {
29              if (i > mid) a[k] = aux[j++];
30              else if (j > hi) a[k] = aux[i++];
31              else if (less(aux[j], aux[i])) a[k] = aux[j++];
32              else a[k] = aux[i++];
33          }
34
35          assert isSorted(a, lo, hi); // postcondition: a[lo..hi] sorted
36      }
37  }
```

4.4.3 Laufzeitanalyse

- **Best-Case:** Der Best-Case tritt auf, wenn die Eingabedaten bereits vollständig sortiert sind. In diesem Fall teilt der Algorithmus die Liste weiterhin in Hälften und führt sie wieder zusammen, wobei die bereits sortierte Reihenfolge beibehalten wird. Die Best-Case-Laufzeit beträgt $O(n \log n)$, wobei n die Anzahl der Elemente in der Liste ist.
- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. Die Average-Case-Laufzeit beträgt ebenfalls $O(n \log n)$, was besser ist als die quadratische Laufzeit von Insertion Sort oder Selection Sort.
- **Worst-Case:** Der Worst-Case tritt auf, wenn die Eingabedaten in umgekehrter Reihenfolge sortiert sind. Der Algorithmus teilt die Liste weiterhin in Hälften und führt sie wieder zusammen, wobei die umgekehrte Reihenfolge in die korrekte Reihenfolge geändert wird. Die Worst-Case-Laufzeit beträgt $O(n \log n)$.

Die kleinstmögliche Anzahl an Vergleichen für jeden Sortieralgorithmus ist $\lg(N!) = O(N \lg N)$. Merge Sort hat genau diese Anzahl an Vergleichen und ist daher optimal. Allerdings ist der Algorithmus nicht optimal in Bezug auf Speichernutzung, da er nicht in-place ist.

Aufgrund seiner effizienten Laufzeit ist Merge Sort für große Datensätze geeignet und eine gute Wahl, wenn Stabilität erforderlich ist.

4.4.4 Eigenschaften

- **Stabilität:** Merge Sort ist ein stabiler Sortieralgorithmus, da gleichwertige Elemente ihre relative Reihenfolge beibehalten. Dies liegt daran, dass bei der Zusammenführung der Teilsequenzen das linke Element gewählt wird, wenn zwei Elemente gleich sind, wodurch ihre ursprüngliche Reihenfolge erhalten bleibt.
- **In-Place:** Merge Sort ist im Allgemeinen nicht in-place, da zusätzlicher Speicherplatz benötigt wird, um die Teilsequenzen während der Zusammenführung zu speichern. Es gibt jedoch in-place Varianten von Merge Sort, die jedoch in der Regel komplexer und weniger effizient sind.
- **Parallelisierbar:** Merge Sort kann parallelisiert werden, da die Teilsequenzen unabhängig voneinander sortiert werden können. Dies kann zu einer weiteren Verbesserung der Leistung führen, insbesondere bei großen Datensätzen und auf Systemen mit mehreren Prozessorkernen.

Insgesamt ist Merge Sort eine gute Wahl für große Datensätze und Anwendungsfälle, in denen Stabilität erforderlich ist. Wenn jedoch zusätzlicher Speicherplatz ein Problem darstellt, können andere effiziente Sortieralgorithmen wie Quick Sort oder Heap Sort in Betracht gezogen werden.

4.5 Quick Sort

Quick Sort ist ein effizienter und in-place Sortieralgorithmus, der auf dem Prinzip des “Teilen und Herrschen” basiert. Der Algorithmus wählt ein Element, das sogenannte Pivot, aus der Liste aus und organisiert die Liste so, dass alle Elemente, die kleiner als das Pivot sind, links davon stehen und alle größeren Elemente rechts davon. Quick Sort eignet sich gut für große Datensätze und zeigt eine bessere Leistung als einfache Sortieralgorithmen wie Insertion Sort oder Selection Sort.

4.5.1 Algorithmus

1. Wähle ein Pivot-Element aus der Liste.
2. Organisiere die Liste, sodass alle Elemente kleiner als das Pivot links davon stehen und alle größeren Elemente rechts davon.
3. Sortiere die linke und rechte Teilsequenz rekursiv mit Quick Sort.
4. Kombiniere die sortierten Teilsequenzen.

4.5.2 Quick Sort mit Cutoff

Ein Cutoff ist eine Optimierung für Quick Sort, bei der für kleine Teilsequenzen auf einen einfacheren Sortieralgorithmus wie Insertion Sort umgeschaltet wird. Dies kann die Leistung verbessern, da Insertion Sort für kleine Datensätze effizienter ist und weniger Funktionsaufrufe benötigt werden.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

4.5.3 Median-of-3

Eine weitere Optimierung für Quick Sort besteht darin, den Median von drei zufällig ausgewählten Elementen als Pivot zu verwenden. Dies verringert die Wahrscheinlichkeit, dass ein schlechtes Pivot gewählt wird, und verbessert die Leistung, insbesondere im Worst-Case-Szenario.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }

    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

4.5.4 Shuffle

Die Shuffle-Optimierung besteht darin, die Liste vor der Sortierung zufällig zu mischen. Dies stellt sicher, dass die Eingabedaten keine besondere Ordnung aufweisen und verhindert (bzw. macht sehr sehr

unwahrscheinlich) den Worst-Case, indem eine gleichmäßige Verteilung der Elemente gewährleistet wird.

4.5.5 Java Implementierung

```java
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    {
        int i = lo, j = hi+1;
        while (true)
        {
            while (less(a[++i], a[lo]))
                if (i == hi) break;

            while (less(a[lo], a[--j]))
                if (j == lo) break;

            if (i >= j) break;
            exch(a, i, j);
        }

        exch(a, lo, j);
        return j;
    }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;

        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

4.5.6 Laufzeitanalyse

- **Best-Case:** Der Best-Case tritt auf, wenn das Pivot bei jedem Schritt das Median-Element der Teilsequenz ist. In diesem Fall teilt der Algorithmus die Liste in gleich große Hälften und erreicht

eine Laufzeit von $O(n \log n)$, wobei n die Anzahl der Elemente in der Liste ist.

- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. Die Average-Case-Laufzeit beträgt ebenfalls $O(n \log n)$.
- **Worst-Case:** Der Worst-Case tritt auf, wenn das Pivot bei jedem Schritt das kleinste oder größte Element der Teilsequenz ist. In diesem Fall teilt der Algorithmus die Liste in sehr ungleiche Teilsequenzen, was zu einer Laufzeit von $O(n^2)$ führt. Durch die Anwendung der Median-of-3-Optimierung kann diese Worst-Case-Laufzeit jedoch reduziert werden, sodass sie in der Praxis seltener auftritt.

4.5.7 Eigenschaften

- **Stabilität:** Quick Sort ist im Allgemeinen nicht stabil, da die Reihenfolge gleichwertiger Elemente während der Partitionierung geändert werden kann. Es gibt jedoch stabile Varianten von Quick Sort, die jedoch oft zusätzlichen Speicherplatz oder Komplexität erfordern.
- **In-Place:** Quick Sort ist in-place, da es keine zusätzlichen Datenstrukturen benötigt und die Sortierung innerhalb der gegebenen Liste durchführt.
- **Parallelisierbar:** Quick Sort kann parallelisiert werden, indem die rekursiven Aufrufe auf separaten Threads oder Prozessorkernen ausgeführt werden. Dies kann die Leistung weiter verbessern, insbesondere bei großen Datensätzen und auf Systemen mit mehreren Prozessorkernen.

Insgesamt ist Quick Sort eine gute Wahl für große Datensätze und kann durch Optimierungen wie Cutoff und Median-of-3 noch weiter verbessert werden. Bei Bedarf an Stabilität sollte jedoch auf alternative Algorithmen wie Merge Sort zurückgegriffen werden.

4.6 Quick Select

Quick Select ist ein Auswahlalgorithmus, der auf dem Prinzip des “Teilen und Herrschen” basiert und eng mit dem Quick Sort-Algorithmus verwandt ist. Der Algorithmus wird verwendet, um das k-te kleinste Element in einer ungeordneten Liste zu finden, ohne die gesamte Liste zu sortieren. Die Laufzeit von Quick Select ist im Durchschnitt besser als die von Sortieralgorithmen wie Quick Sort oder Merge Sort, da nur ein Teil der Liste bearbeitet werden muss.

4.6.1 Algorithmus

1. Wähle ein Pivot-Element aus der Liste.

2. Organisiere die Liste, sodass alle Elemente kleiner als das Pivot links davon stehen und alle größeren Elemente rechts davon.
3. Wenn der Pivot-Index gleich k ist, wurde das k-te Element gefunden. Andernfalls suche rekursiv in der linken oder rechten Teilsequenz, je nachdem, ob k kleiner oder größer als der Pivot-Index ist.

4.6.2 Java Implementierung

```java
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
        int j = partition(a, lo, hi);
        if (j < k) lo = j + 1;
        else if (j > k) hi = j - 1;
        else return a[k];
    }
    return a[k];
}
```

4.6.3 Laufzeitanalyse

- **Best-Case:** Der Best-Case tritt auf, wenn das Pivot bei jedem Schritt das Median-Element der Teilsequenz ist. In diesem Fall teilt der Algorithmus die Liste in gleich große Hälften und erreicht eine Laufzeit von $O(n)$, wobei n die Anzahl der Elemente in der Liste ist.
- **Average-Case:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. Die Average-Case-Laufzeit beträgt ebenfalls $O(n)$.
- **Worst-Case:** Der Worst-Case tritt auf, wenn das Pivot bei jedem Schritt das kleinste oder größte Element der Teilsequenz ist. In diesem Fall teilt der Algorithmus die Liste in sehr ungleiche Teilsequenzen, was zu einer Laufzeit von $O(n^2)$ führt. Durch die Verwendung von Optimierungen wie Median-of-3 kann die Wahrscheinlichkeit des Worst-Case-Szenarios jedoch reduziert werden.

Quick Select ist eine effiziente Methode, um das k-te kleinste Element in einer Liste zu finden, insbesondere bei großen Datensätzen. Durch die Verwendung von Optimierungen wie Median-of-3 kann die Laufzeit des Algorithmus weiter verbessert und das Auftreten von Worst-Case-Szenarien reduziert werden. Es ist jedoch wichtig zu beachten, dass Quick Select nicht zum Sortieren der gesamten Liste

verwendet wird. Wenn das Ziel darin besteht, die Liste vollständig zu sortieren, sind Sortieralgorithmen wie Quick Sort oder Merge Sort besser geeignet.

4.7 3-Way Quick Sort

3-Way Quick Sort ist eine Variante des klassischen Quick Sort Algorithmus, die besonders gut mit Datensätzen umgehen kann, die viele gleiche Elemente enthalten. Im Gegensatz zum herkömmlichen Quick Sort, der die Liste in zwei Teilsequenzen (kleiner als das Pivot und größer als das Pivot) aufteilt, teilt 3-Way Quick Sort die Liste in drei Teilsequenzen auf: Elemente kleiner als das Pivot, Elemente gleich dem Pivot und Elemente größer als das Pivot.

4.7.1 Algorithmus

1. Wähle ein Pivot-Element aus der Liste.
2. Organisiere die Liste in drei Teilsequenzen: Elemente kleiner als das Pivot, Elemente gleich dem Pivot und Elemente größer als das Pivot.
3. Sortiere die linke und rechte Teilsequenz rekursiv mit 3-Way Quick Sort.
4. Kombiniere die sortierten Teilsequenzen.

Der Hauptvorteil von 3-Way Quick Sort besteht darin, dass gleiche Elemente nicht weiter sortiert werden müssen, da sie bereits in der Mitte der Liste gruppiert sind. Dies kann die Leistung des Algorithmus insbesondere bei Datensätzen mit vielen gleichen Elementen verbessern.

Die Laufzeitanalyse von 3-Way Quick Sort ähnelt der des klassischen Quick Sort. Die Laufzeit für Best-, Average- und Worst-Case bleibt gleich. In der Praxis ist 3-Way Quick Sort aber besser, wenn es wenige einzigartige Elemente gibt.

Die 3-Way Quick Sort Variante ist ebenfalls in-place und nicht stabil.

4.8 Heap Sort

Heap Sort ist ein effizienter, in-place und vergleichsbasierter Sortieralgorithmus, der auf der Heap-Datenstruktur basiert. Heap Sort verwendet die Struktur und die Eigenschaften eines Max-Heaps, um die Elemente einer Liste oder eines Arrays in aufsteigender Reihenfolge zu sortieren. Die Laufzeit von HeapSort beträgt $O(n \log n)$, wobei n die Anzahl der Elemente in der Liste ist.

4.8.1 Algorithmus

1. Erstelle einen Max-Heap aus den zu sortierenden Elementen.

2. Entferne das Element mit der höchsten Priorität (Wurzel des Max-Heaps) und tausche es mit dem letzten Element im Heap.
3. Verringere die Größe des Heaps um eins und stelle die Heap-Eigenschaft für das neue Wurzelelement wieder her (sink).
4. Wiederhole Schritt 2 und 3, bis der Heap leer ist.

4.8.2 Java Implementierung

```java
public class HeapSort {
    public static void sort(Comparable[] a) {
        int n = a.length;

        // Erstelle Max-Heap
        for (int k = n / 2; k >= 1; k--) {
            sink(a, k, n);
        }

        // Sortiere Elemente
        while (n > 1) {
            swap(a, 1, n--);
            sink(a, 1, n);
        }
    }

    private static void sink(Comparable[] a, int k, int n) {
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && less(a, j, j + 1)) j++;
            if (!less(a, k, j)) break;
            swap(a, k, j);
            k = j;
        }
    }

    // Hilfsmethoden (less und swap)...
}
```

4.8.3 Laufzeitanalyse

HeapSort hat eine Laufzeit von $O(n \log n)$, da das Erstellen des Max-Heaps eine Laufzeit von $O(n)$ hat und das Entfernen von Elementen aus dem Heap eine Laufzeit von $O(\log n)$ pro Element hat. Insgesamt führt dies zu einer Laufzeit von $O(n) + n * O(\log n) = O(n \log n)$. Dies ist eine gute Laufzeit für vergleichsbasierte Sortieralgorithmen und vergleichbar mit anderen effizienten Algorithmen wie QuickSort und MergeSort.

4.8.4 Eigenschaften

- **In-Place:** HeapSort ist ein in-place-Algorithmus, da er keine zusätzlichen Datenstrukturen benötigt und die Sortierung innerhalb der gegebenen Liste oder des Arrays durchführt. Im Vergleich zu MergeSort, das zusätzlichen Speicherplatz benötigt, ist HeapSort speichereffizienter.
- **Stabilität:** HeapSort ist im Allgemeinen nicht stabil, da die Reihenfolge von Elementen mit gleicher Priorität während des Sinkens geändert werden kann. Wenn Stabilität erforderlich ist, sollten alternative Sortieralgorithmen wie MergeSort verwendet werden.
- **Parallelisierbar:** HeapSort ist nicht parallelisierbar, da es auf der sequenziellen Manipulation eines Heaps basiert. Divide & Conquer Algorithmen, wie Quick Sort oder Merge Sort sind in der Hinsicht besser.

4.8.5 Vergleich mit anderen Sortieralgorithmen

- **QuickSort:** QuickSort hat im Durchschnitt eine ähnliche Laufzeit wie HeapSort ($O(n \log n)$), aber QuickSort ist oft schneller in der Praxis aufgrund besserer Cache-Effizienz. QuickSort hat jedoch eine Worst-Case-Laufzeit von $O(n^2)$, die durch Optimierungen wie Median-of-3 reduziert werden kann. HeapSort hat eine garantierte Laufzeit von $O(n \log n)$, ist jedoch nicht stabil.
- **MergeSort:** MergeSort hat ebenfalls eine Laufzeit von $O(n \log n)$ und ist stabil, was es zu einer guten Wahl für den Umgang mit gleichwertigen Elementen macht. MergeSort ist jedoch nicht in-place und benötigt zusätzlichen Speicherplatz, während HeapSort in-place ist.

Insgesamt ist HeapSort eine gute Wahl für einen effizienten, in-place Sortieralgorithmus, wenn Stabilität keine Anforderung ist. HeapSort hat eine garantierte Laufzeit von $O(n \log n)$ und kann in vielen Anwendungsfällen eine gute Leistung bieten.

5 Suchalgorithmen

5.1 Sequentielle Suche

Sequentielle Suche, auch bekannt als lineare Suche, ist eine einfache Suchmethode, die darauf abzielt, ein bestimmtes Element in einer Liste oder einem Array zu finden, indem jedes Element in der Liste der Reihe nach durchsucht wird. Sequentielle Suche ist im Allgemeinen langsamer als andere Suchmethoden wie binäre Suche oder Hashing, hat jedoch den Vorteil, dass sie auf unsortierten Daten und Datenstrukturen mit dynamischer Größe angewendet werden kann.

5.1.1 Java Implementierung

```java
public class SequentialSearch {
    public static int search(Comparable[] a, Comparable key) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].compareTo(key) == 0) {
                return i;
            }
        }
        return -1;
    }
}
```

5.1.2 Kostenanalyse

5.1.2.1 Search (Suche):

- Worst-Case: Die Worst-Case-Kosten treten auf, wenn das gesuchte Element am Ende der Liste steht oder nicht in der Liste enthalten ist. In diesem Fall beträgt die Laufzeit $O(n)$, wobei n die Anzahl der Elemente in der Liste ist.
- Average-Case: Im Durchschnitt, wenn die Elemente gleichmäßig verteilt sind, beträgt die Laufzeit der Suche $O(\frac{n}{2})$, da im Mittel die Hälfte der Elemente überprüft werden muss, um das gesuchte Element zu finden.

5.1.2.2 Insert (Einfügen):

- Worst-Case: Die Worst-Case-Kosten für das Einfügen treten auf, wenn das Element bereits in der Liste vorhanden ist. In diesem Fall beträgt die Laufzeit $O(n)$, da das gesamte Array durchlaufen werden muss, um das Element zu finden und den Wert zu aktualisieren.
- Average-Case: Die durchschnittlichen Kosten für das Einfügen betragen ebenfalls $O(n)$, da wir durch die gesamte Liste laufen müssen, um den richtigen Platz für das neue Element zu finden.

5.1.2.3 Delete (Löschen):

- Worst-Case: Die Worst-Case-Kosten für das Löschen treten auf, wenn das zu löschende Element am Ende der Liste steht. In diesem Fall beträgt die Laufzeit $O(n)$, da das gesamte Array durchlaufen werden muss, um das Element zu finden und zu entfernen.
- Average-Case: Im Durchschnitt beträgt die Laufzeit des Löschens $O(\frac{n}{2})$, da im Mittel die Hälfte der Elemente überprüft werden muss, um das zu löschende Element zu finden und zu entfernen.

5.2 Binäre Suche

Die binäre Suche ist ein Suchalgorithmus, der auf sortierten Listen oder Arrays basiert und durch wiederholtes Halbieren des Suchintervalls effizient nach einem bestimmten Element sucht. Binäre Suche eignet sich gut für große Datensätze und zeigt eine bessere Leistung als einfache Suchalgorithmen wie die lineare Suche.

5.2.1 Algorithmus

1. Beginne mit dem gesamten sortierten Array oder der Liste.
2. Vergleiche das mittlere Element mit dem gesuchten Wert.
3. Wenn das mittlere Element gleich dem gesuchten Wert ist, ist die Suche erfolgreich.
4. Wenn das mittlere Element kleiner als der gesuchte Wert ist, wiederhole die Suche im rechten Teil der Liste.
5. Wenn das mittlere Element größer als der gesuchte Wert ist, wiederhole die Suche im linken Teil der Liste.
6. Wenn der gesuchte Wert nicht gefunden wird und das Suchintervall leer ist, ist die Suche erfolglos.

5.2.2 Java Implementierung

```java
public class BinarySearch {

    public static int search(Comparable[] a, Comparable key) {
        int lo = 0;
        int hi = a.length - 1;

        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(a[mid]);

            if (cmp < 0) {
                hi = mid - 1;
            } else if (cmp > 0) {
                lo = mid + 1;
            } else {
                return mid;
            }
        }

        return -1;
    }
}
```

5.2.3 Laufzeitanalyse

- **Worst-Case Suche:** Die Worst-Case-Laufzeit für die binäre Suche beträgt $O(\log n)$, wobei n die Anzahl der Elemente in der Liste ist. Im Worst-Case muss der Algorithmus das gesamte Array oder die Liste durchsuchen, um den gesuchten Wert zu finden.
- **Average-Case Suche:** Im Durchschnitt wird angenommen, dass die Eingabedaten keine besondere Ordnung aufweisen und gleichmäßig verteilt sind. Die Average-Case-Laufzeit für die Suche beträgt ebenfalls $O(\log n)$.

Die Analyse von Insert und Delete Operationen ist schwieriger, da sie stark von der zugrunde liegenden Datenstruktur abhängen. Bei Verwendung eines Arrays sind die Worst-Case-Kosten für das Einfügen und Löschen von Elementen $O(n)$, da im schlimmsten Fall die Elemente verschoben werden müssen, um Platz für das neue Element zu schaffen oder das gelöschte Element zu entfernen. Für durchschnittliche Fälle können diese Kosten jedoch reduziert werden, insbesondere wenn eine geeignete Datenstruktur wie ein balancierter Suchbaum verwendet wird.

5.2.4 Eigenschaften

- **Effizienz:** Die binäre Suche ist sehr effizient, speziell bei großen Datensätzen, da sie die Anzahl der Vergleiche im Vergleich zur linearen Suche drastisch reduziert.
- **Voraussetzung:** Die binäre Suche erfordert, dass die Liste oder das Array im Voraus sortiert ist. Wenn dies nicht der Fall ist, müssen die Daten zunächst sortiert werden, was zusätzliche Kosten verursacht.

Insgesamt ist die binäre Suche ein schneller und effizienter Algorithmus

5.3 Binärer Suchbaum

Ein binärer Suchbaum (BST) ist eine hierarchische Datenstruktur, die auf Knoten basiert und die Eigenschaft hat, dass jeder Knoten höchstens zwei Kindknoten besitzt. Ein binärer Suchbaum erlaubt schnelle Suche, Einfügung und Löschung von Elementen, indem er die binäre Suche auf der Baumstruktur anwendet. In einem binären Suchbaum gilt, dass alle Elemente im linken Teilbaum eines Knotens kleiner als der Wert des Knotens sind und alle Elemente im rechten Teilbaum größer als der Wert des Knotens sind.

5.3.1 Algorithmus

5.3.1.1 Suche

1. Beginne bei der Wurzel des Baums.
2. Vergleiche den gesuchten Wert mit dem Wert des aktuellen Knotens.
3. Wenn der gesuchte Wert kleiner ist, gehe zum linken Kindknoten; wenn der gesuchte Wert größer ist, gehe zum rechten Kindknoten.
4. Wiederhole den Vorgang, bis der gesuchte Wert gefunden wird oder kein weiterer Kindknoten vorhanden ist.

5.3.1.2 Einfügen

1. Beginne bei der Wurzel des Baums.
2. Vergleiche den einzufügenden Wert mit dem Wert des aktuellen Knotens.
3. Wenn der einzufügende Wert kleiner ist, gehe zum linken Kindknoten; wenn der einzufügende Wert größer ist, gehe zum rechten Kindknoten.
4. Wiederhole den Vorgang, bis ein leerer Platz gefunden wird, und füge den neuen Wert als Kindknoten ein.

5.3.1.3 Löschen

1. Suche den Knoten mit dem zu löschenden Wert.
2. Wenn der Knoten keine Kindknoten hat, entferne einfach den Knoten.
3. Wenn der Knoten nur ein Kind hat, entferne den Knoten und ersetze ihn durch seinen Kindknoten.
4. Wenn der Knoten zwei Kindknoten hat, finde den nächstgrößeren Wert im Unterbaum (also dem Knoten folgenden Knoten), ersetze den Knotenwert durch diesen Wert und entferne den Knoten mit dem ersetzten Wert.

5.3.2 Java Implementierung

```java
public class BinarySearchTree<Key extends Comparable<Key>, Value> {
    private Node root;

    private class Node {
        private Key key;
        private Value value;
        private Node left, right;
        private int count;

        public Node(Key key, Value value) {
            this.key = key;
            this.value = value;
            this.count = 1;
        }
    }
```

```
16
17      public Value search(Key key) {
18          Node x = search(root, key);
19          if (x == null) return null;
20          return x.value;
21      }
22
23      private Node search(Node x, Key key) {
24          if (x == null) return null;
25          int cmp = key.compareTo(x.key);
26          if (cmp < 0) return search(x.left, key);
27          else if (cmp > 0) return search(x.right, key);
28          else return x;
29      }
30
31      public void insert(Key key, Value value) {
32          root = insert(root, key, value);
33      }
34
35      private Node insert(Node x, Key key, Value value) {
36          if (x == null) return new Node(key, value);
37          int cmp = key.compareTo(x.key);
38          if (cmp < 0) x.left = insert(x.left, key, value);
39          else if (cmp > 0) x.right = insert(x.right, key, value);
40          else x.value = value;
41          x.count = 1 + size(x.left) + size(x.right);
42          return x;
43      }
44
45      public void delete(Key key) {
46          root = delete(root, key);
47      }
48
49      private Node delete(Node x, Key key) {
50          if (x == null) return null;
51
52          int cmp = key.compareTo(x.key);
53          if (cmp < 0) x.left = delete(x.left, key);
54          else if (cmp > 0) x.right = delete(x.right, key);
55          else {
56              if (x.right == null) return x.left;
57              if (x.left == null) return x.right;
58
59              Node t = x;
60              x = min(t.right);
61              x.right = deleteMin(t.right);
62              x.left = t.left;
63          }
64
65          x.count = size(x.left) + size(x.right) + 1;
66          return x;
```

```
67      }
68
69      private Node min(Node x) {
70          if (x.left == null) return x;
71          return min(x.left);
72      }
73
74      private Node deleteMin(Node x) {
75          if (x.left == null) return x.right;
76          x.left = deleteMin(x.left);
77          x.count = size(x.left) + size(x.right) + 1;
78          return x;
79      }
80
81      private int size(Node x) {
82          if (x == null) return 0;
83          return x.count;
84      }
85
86  }
```

5.3.3 Laufzeitanalyse

- **Suche:** Die Worst-Case-Laufzeit für die Suche in einem binären Suchbaum beträgt $O(h)$, wobei h die Höhe des Baums ist. Im Worst-Case kann der Baum vollständig unbalanciert sein und einer linearen Liste ähneln.
- **Einfügen:** Die Worst-Case-Laufzeit für das Einfügen in einem binären Suchbaum beträgt ebenfalls $O(h)$. Im schlimmsten Fall muss der Algorithmus bis zum tiefsten Knoten navigieren, um das neue Element einzufügen.
- **Löschen:** Die Worst-Case-Laufzeit für das Löschen in einem binären Suchbaum beträgt $O(h)$, da im schlimmsten Fall ein Knoten am unteren Rand des Baums gelöscht werden muss.

Für einen balancierten binären Suchbaum beträgt die Höhe des Baums $O(\log n)$, wobei n die Anzahl der Elemente im Baum ist. In diesem Fall sind die durchschnittlichen Kosten für Suche, Einfügung und Löschung ebenfalls $O(\log n)$.

5.3.4 Eigenschaften

- **Effizienz:** Binäre Suchbäume bieten eine effiziente Implementierung von Such-, Einfüge- und Löschoperationen, insbesondere wenn der Baum gut ausbalanciert ist.

- **Balancierung:** Um die Effizienz des Baums zu gewährleisten, ist es wichtig, den binären Suchbaum ausbalanciert zu halten. Eine Möglichkeit, dies zu erreichen, besteht darin, selbstbalancierende binäre Suchbäume wie Rot-Schwarz-Bäume zu verwenden.
- **Sortierung:** Da die Elemente im binären Suchbaum auf einer Inorder-Traversierung in sortierter Reihenfolge erscheinen, kann der Baum auch zum Sortieren von Elementen verwendet werden.

Insgesamt bieten binäre Suchbäume eine effiziente und strukturierte Möglichkeit, um Such-, Einfüge- und Löschoperationen durchzuführen. Durch die Verwendung von selbstbalancierenden Varianten können die durchschnittlichen Kosten der Operationen auf $O(\log n)$ reduziert werden, wodurch sie sich besonders gut für dynamische Datenmengen eignen, bei denen häufige Änderungen erforderlich sind.

5.4 2-3 Bäume

Ein 2-3 Baum ist eine spezielle Art von balanciertem Suchbaum, bei dem jeder Knoten entweder zwei oder drei Kinder haben kann. Dieser Baum gewährleistet eine gute Balance und führt zu einer effizienten Suche, Einfügung und Löschung von Elementen.

5.4.1 Struktur

Ein Knoten im 2-3 Baum kann zwei verschiedene Arten von Knoten sein:

1. **2-Knoten:** Ein 2-Knoten hat genau einen Schlüssel und zwei Kindknoten. Der linke Kindknoten enthält Elemente, die kleiner als der Schlüssel sind, und der rechte Kindknoten enthält Elemente, die größer als der Schlüssel sind.
2. **3-Knoten:** Ein 3-Knoten hat zwei Schlüssel und drei Kindknoten. Der linke Kindknoten enthält Elemente, die kleiner als der erste Schlüssel sind, der mittlere Kindknoten enthält Elemente, die zwischen den beiden Schlüsseln liegen, und der rechte Kindknoten enthält Elemente, die größer als der zweite Schlüssel sind.

Alle Blätter im 2-3 Baum haben dieselbe Tiefe, was bedeutet, dass der Baum perfekt ausbalanciert ist.

5.4.2 Algorithmus

5.4.2.1 Suche Die Suche in einem 2-3 Baum ist ähnlich wie die Suche in einem binären Suchbaum. Beginne an der Wurzel und gehe entlang der Knoten, bis der gesuchte Schlüssel gefunden wird oder ein Blatt erreicht wird.

5.4.2.2 Einfügen Um einen Schlüssel in einen 2-3 Baum einzufügen, führe die folgenden Schritte aus:

1. Beginne an der Wurzel und gehe entlang der Knoten, bis ein Blatt erreicht wird.
2. Füge den Schlüssel in den entsprechenden Knoten ein. Wenn der Knoten ein 2-Knoten ist, wird er zu einem 3-Knoten. Wenn der Knoten bereits ein 3-Knoten ist, wird er zu einem temporären 4-Knoten.
3. Wenn ein 4-Knoten entstanden ist, teile den Knoten auf und schiebe den mittleren Schlüssel nach oben in den Elternknoten. Wenn der Elternknoten ebenfalls ein 4-Knoten wird, wiederhole den Vorgang.
4. Wenn die Wurzel zu einem 4-Knoten wird, teile die Wurzel auf und erstelle eine neue Wurzel mit dem mittleren Schlüssel. Die Höhe des Baums erhöht sich um 1.

5.4.2.3 Löschen Um einen Schlüssel aus einem 2-3 Baum zu löschen, führe die folgenden Schritte aus:

1. Finde den Knoten, der den Schlüssel enthält.
2. Wenn der Knoten ein Blatt ist, entferne einfach den Schlüssel aus dem Knoten. Andernfalls finde den Nachfolger oder Vorgänger des Schlüssels im Baum, tausche die Positionen und entferne den Schlüssel aus dem Blattknoten.
3. Falls notwendig, passe den Baum an, um die 2-3 Baum-Eigenschaften aufrechtzuerhalten. Wenn der Knoten, aus dem der Schlüssel entfernt wurde, jetzt ein 1-Knoten (ein Knoten ohne Schlüssel) ist, müssen die folgenden Anpassungen vorgenommen werden:
    - **Fall 1:** Wenn ein benachbarter Geschwisterknoten ein 3-Knoten ist, rotiere den Baum, um den 1-Knoten aufzufüllen und einen Schlüssel vom benachbarten Geschwisterknoten zu übernehmen.
    - **Fall 2:** Wenn alle Geschwisterknoten 2-Knoten sind, führe den 1-Knoten mit einem Geschwisterknoten zusammen und verschiebe einen Schlüssel vom Elternknoten nach unten in den neu fusionierten Knoten. Wenn der Elternknoten dadurch zu einem 1-Knoten wird, wende die Anpassungen rekursiv an.
    - **Fall 3:** Wenn die Wurzel ein 1-Knoten wird und nur zwei Kindknoten hat, entferne die Wurzel und mache das verbleibende Kind zur neuen Wurzel. Die Höhe des Baums verringert sich um 1.

5.4.3 Laufzeitanalyse

Da alle Blätter in einem 2-3 Baum dieselbe Tiefe haben und der Baum perfekt ausbalanciert ist, beträgt die Höhe des Baums $O(\log n)$, wobei n die Anzahl der Elemente im Baum ist. Daher sind die Worst-Case-Kosten für Suche, Einfügen und Löschen von Elementen alle $O(\log n)$.

5.4.4 Eigenschaften

- **Effizienz:** 2-3 Bäume bieten eine effiziente Implementierung von Such-, Einfüge- und Löschoperationen, da sie aufgrund ihrer perfekt ausbalancierten Struktur konstante Höhe haben.
- **Anpassungsfähigkeit:** 2-3 Bäume sind gut anpassungsfähig und eignen sich für Anwendungen, bei denen häufige Änderungen an der Datenstruktur erforderlich sind.

Insgesamt bieten 2-3 Bäume eine gute Balance aus Effizienz und einfacher Implementierung für Such-, Einfüge- und Löschoperationen. Sie sind eine gute Wahl für Anwendungen, bei denen eine ausbalancierte Datenstruktur erforderlich ist und die Komplexität der Implementierung minimiert werden soll.

5.4.5 Implementierung

Eine leichte Implementierung sind Schwarz-Rot-Bäume. Die Idee ist, von einem 2-Node BST auszugehen und jedem Link entweder rot oder schwarz zu markieren. Jede 3-Node wird dargestellt als zwei 2-Nodes, welche durch einen roten Link verbunden sind. Die obere 2-Node erhält den größeren Schlüssel und das größte Kindknoten, während die untere 2-Node den kleineren Schlüssel und die 2 kleineren Kindknoten bekommt.

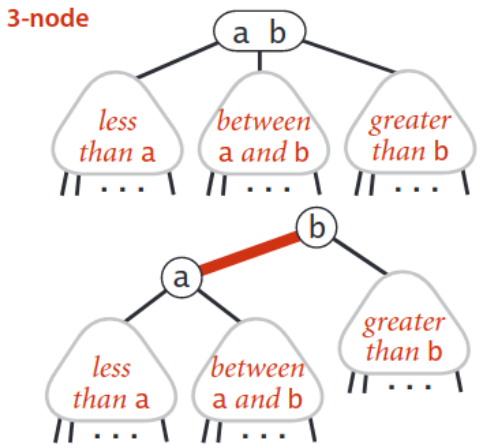


Abbildung 12: Überführen einer 3-Node zu zwei 2-Nodes

Eine äquivalente Definition ist durch die folgenden 3 Regeln gegeben:

- Rote Links sind links
- Keine Node hat zwei rote Links
- Der Baum hat perfekte Schwarz-Balance. Das heißt, jeder Pfad von der Wurzel aus hat die gleiche Anzahl an roten Links.

Zeichnet man die roten Links horizontal, dann lässt sich direkt die Struktur des 2-3-Baums erkennen. Da ein Schwarz-Rot-Baum ein BST mit zusätzlichen Informationen ist, können diese Informationen ignoriert werden, wodurch der Baum gleichzeitig als 2-Node BST gelesen werden kann.

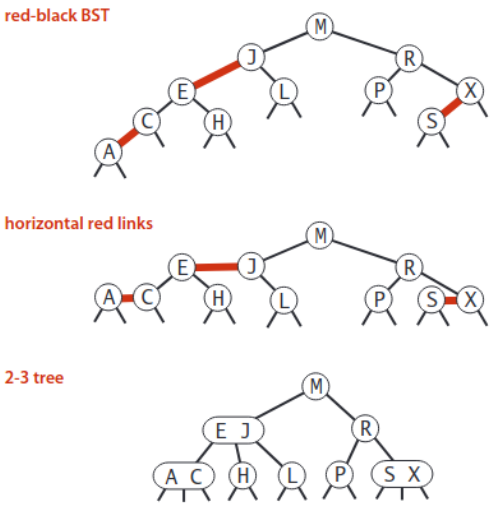


Abbildung 13: Gleichheit des Rot-Schwarz-BSTs und 2-3-Baums

Für Code und Beispiel, siehe im Buch auf S. 440.

5.5 Suchbäume Laufzeit

implementation	worst-case cost (after N inserts)			average-case cost (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	**equals()**
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	**compareTo()**
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	**compareTo()**
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	**compareTo()**

6 Hashing

Alle Elemente werden in einer key-indexed Tabelle abgespeichert. Der Index wird basierend auf dem Schlüssel mit einer Hash-Funktion berechnet. Das funktioniert aber nur so lange gut, bis zwei Schlüssel den gleichen Index zugewiesen bekommen. Dies nennt man eine Kollision und in den folgenden Abschnitten geht es um Algorithmen und Datenstrukturen, die die Handhabung von Kollisionen ermöglichen.

Hashing ist ein Trade-off zwischen Platz und Zeit. Gibt es keine Platzeinschränkungen, dann kann der Schlüssel selber als Index genutzt werden. Gibt es keine Zeitbeschränkung, kann die Kollision mit sequenzieller Suche erkannt werden. Hashing ist dazwischen und bietet eine gute Balance zwischen Platz und Zeit.

6.1 Hashfunktion

Die ideale Hashfunktion ist effizient berechenbar und hat für jeden Index die gleiche Wahrscheinlichkeit, wodurch die Schlüssel gleichmäßig verteilt werden.

Diese ideale Hashfunktion wird oft als Voraussetzung genommen. Das nennt man dann die **Uniform Hashing Assumption**. In der Uniform Hashing Assumption steht M für die Größe des Hasharrays und N für die Anzahl an Schlüsseln, die gehasht werden.

6.1.1 Java

In Java hat jede eingebaute Klasse die Funktion `x.hashCode()`. Dabei gilt, wenn `x.equals(y)`, dann `x.hashCode()==y.hashCode()`.

Hier sind ein paar Implementierungen für eingebaute Datentypen:

- **Int**

```java
public final class Integer
{
    private final int value;
    ...

    public int hashCode()
    { return value; }
}
```

- **Boolean**

```java
public final class Boolean
{
    private final boolean value;
    ...

    public int hashCode()
    {
        if (value) return 1231;
        else return 1237;
    }
}
```

- **String**

```java
public final class String
{
    private final char[] s;
    ...

    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

Java schlägt für die Implementierung einer Hash-Funktion bei einer benutzerdefinierten Klasse folgende Regeln vor:

- Kombinieren jedes signifikante Feld mit der 31x + y Regel.
- Wenn das Feld ein primitiver Typ ist, verwende die Wrapper-Typ hashCode() Funktion.
- Wenn das Feld null ist, gebe 0 zurück.
- Wenn das Feld ein Referenztyp ist, verwende die `hashCode()` Funktion.
- Wenn das Feld ein Array ist, wende dies auf jeden Eintrag an.

6.2 Seperate Chaining

Die Idee hinter Hashing mit Seperate Chaining ist bei jedem Index eine Linked-List zu erstellen, wobei jeder Eintrag den Schlüssel und den Wert hält.

Unter der Uniform Hashing Assumption, ist die Anzahl an Elementen in jeder Linked-List gleich N/M. Da die Anzahl an nötigen Operationen für Search / Insert proportional zu der Anzahl an Elementen in der Linked-List ist, hat die Wahl von M eine wichtige Rolle:

- Ist M zu groß, dann hat das Array viele leere Linked-Lists
- Ist M zu klein, dann werden die Linked-Lists zu lang

Daher wählt man meistens $M = N/5$.

Seperate Chaing ist einfach zu implementieren und weniger anfällig gegen Clustering.

6.2.1 Two-probe Hashing

Anstatt nur eine Hashing-Funktion zu nutzen, werden mithilfe von zwei Funktionen zwei Positionen für einen Key berechnet. Der Schlüssel wird dann bei der kürzeren der beiden Ketten eingefügt. Das reduziert die erwartete Länge der längsten Kette auf $\log \log N$.

6.3 Linear Probing

Bei Linear Probing wird bei einer Kollision der nächst leere Platz gesucht und gefüllt.

Ein großes Problem, mit dem das lineare Sondieren konfrontiert ist, ist das Clustering. Clustering ist die Tendenz, dass Kollisionen zu langen Reihen von gefüllten Slots führen, was die Effizienz der Hash-Tabelle verringert.

Gegeben ist die Fülle von M als $\alpha = \frac{N}{M}$. Unter der Uniform Hashing Assumption ist die erwartete Zeit für

- Die Suche: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
- Die erfolglose Suche und Einfügen: $\frac{1}{2}(1 + \frac{1}{1-\alpha^2})$

Ist M zu groß, dann gibt es zu viele leere Einträge. Ist M zu klein, dann nimmt die Laufzeit schnell zu. Daher wählt man typischerweise $M = 2N$.

Linear Probing verschwendet weniger Speicherplatz als Seperate Chaining und hat bessere Cache-Perfomance.

6.4 Laufzeit Zusammenfassung

Unter der Annahme einer gleichmäßigen Hash-Funktion gelten folgende Laufzeiten:

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	**equals()**
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	**compareTo()**
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	**compareTo()**
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	**compareTo()**
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	**equals()**
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	**equals()**

7 Graphentheorie

Die Graphentheorie ist ein Bereich der Mathematik, der sich auf die Untersuchung von Graphen konzentriert. Graphen sind mathematische Strukturen, die zur Darstellung von Paarbeziehungen zwischen Objekten verwendet werden. Ein Graph besteht aus Knoten (oder Ecken) und Kanten (oder Linien), die diese Knoten verbinden. Es gibt zwei Haupttypen von Graphen: ungerichtete und gerichtete Graphen.

7.1 Ungerichtete Graphen

Ein ungerichteter Graph ist eine Sammlung von Knoten, die durch Kanten paarweise verbunden sind. Hier hat jede Kante keine Richtung, d.h., sie verbindet zwei Knoten ohne eine spezifische Richtung. In der Informatik und diskreten Mathematik spielen ungerichtete Graphen eine zentrale Rolle, da sie zur Modellierung einer Vielzahl von Problemen und Systemen verwendet werden können.

7.1.1 Anwendungen von ungerichteten Graphen

Ungerichtete Graphen finden breite Anwendung in verschiedenen Bereichen wie Kommunikation, Schaltkreise, Mechanik, Finanzen, Transport, Internet, Spiele, soziale Beziehungen, neuronale Netzwerke, Proteinnetzwerke und chemische Verbindungen. In diesen Kontexten repräsentieren die Knoten und Kanten jeweils unterschiedliche Entitäten und Beziehungen.

7.1.2 Graph API für ungerichtete Graphen

Eine mögliche Implementierung einer Graph API in Java könnte folgendermaßen aussehen:

```java
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj;

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w)
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v)
    { return adj[v]; }
}
```

7.2 Gerichtete Graphen (Digraphs)

Im Gegensatz zu ungerichteten Graphen haben die Kanten in einem gerichteten Graphen eine spezifische Richtung. Das bedeutet, dass jede Kante von einem Knoten zu einem anderen Knoten zeigt. Gerichtete Graphen werden oft verwendet, um Beziehungen darzustellen, die asymmetrisch sind, d.h., Beziehungen, die nicht unbedingt in beide Richtungen gelten.

7.2.1 Anwendungen von gerichteten Graphen

Gerichtete Graphen haben eine breite Palette von Anwendungen in verschiedenen Bereichen, einschließlich Verkehr, Web Navigation, Ökologie, Finanzen, Telekommunikation und Spieltheorie.

7.2.2 Graph API für gerichtete Graphen

Die Implementierung von gerichteten Graphen in Java ist im Prinzip gleich. Der einzige Unterschied, ist, dass bei `addEdge(int v, int w)` nur `adj[v].add(w)` aufgerufen wird.

Eine mögliche Implementierung einer Graph API für gerichtete Graphen könnte folgendermaßen aussehen:

```
1  public class Digraph {
2      Digraph(int V) { /* Erstelle einen leeren Digraph mit V Knoten */ }
3      Digraph(In in) { /* Erstelle einen Digraph aus einem Eingabestrom
           */ }
4      void addEdge(int v, int w) { /* Füge eine gerichtete Kante von v
           nach w hinzu */ }
5      Iterable adj(int v) { /* Gib die Knoten zurück, zu denen ein Pfeil
           von v führt */ }
6      int V() { /* Anzahl der Knoten */ }
7      int E() { /* Anzahl der Kanten */ }
8      String toString() { /* String Darstellung des Digraphen */ }
9      Digraph reverse() { /* Erstelle einen neuen Digraph, der die
           Umkehrung (alle Kanten umgedreht) dieses Digraphen ist */ }
10 }
```

7.3 Gewichtete gerichtete Graphen

Im Rahmen von gewichteten gerichteten Graphen, auch bekannt als gewichtete Digraphen, hat jede Kante zusätzlich zu ihrer Richtung auch ein Gewicht. Dieses Gewicht kann eine Vielzahl von Bedeutungen haben, abhängig von dem Kontext, in dem der Graph verwendet wird. Beispielsweise kann es die Entfernung zwischen zwei Punkten auf einer Karte, die Zeit, die benötigt wird, um von einem Punkt zu einem anderen zu gelangen, oder die Kosten für die Verbindung zwischen zwei Punkten in einem Netzwerk repräsentieren.

7.3.1 Anwendungen von gewichteten gerichteten Graphen

Gewichtete gerichtete Graphen finden Anwendung in vielen Bereichen, wie Verkehrssystemen (um beispielsweise den kürzesten Weg zwischen zwei Punkten zu finden), in der Telekommunikation (um das optimale Routing von Datenpaketen zu bestimmen), in der Betriebsforschung (für Probleme wie das Reisende-Verkäufer-Problem), in der Computernetzwerkplanung und in der Spieltheorie.

7.3.2 Graph API für gewichtete gerichtete Graphen

Die Implementierung von gewichteten gerichteten Graphen erfordert eine kleine Änderung an der API, um die Gewichte zu speichern. Eine mögliche Implementierung könnte folgendermaßen aussehen:

```
1  public class EdgeWeightedDigraph {
2      EdgeWeightedDigraph(int V) { /* Erstelle einen leeren gewichteten
           Digraph mit V Knoten */ }
```

```
3       EdgeWeightedDigraph(In in) { /* Erstelle einen gewichteten Digraph
            aus einem Eingabestrom */ }
4       void addEdge(DirectedEdge e) { /* Füge eine gewichtete gerichtete
            Kante hinzu */ }
5       Iterable adj(int v) { /* Gib die Kanten zurück, zu denen ein Pfeil
            von v führt */ }
6       int V() { /* Anzahl der Knoten */ }
7       int E() { /* Anzahl der Kanten */ }
8       String toString() { /* String Darstellung des gewichteten Digraphen
             */ }
9       Iterable edges() { /* Gib alle Kanten in diesem Digraphen zurück */
             }
10  }
```

Die Klasse `DirectedEdge` repräsentiert eine gewichtete Kante und könnte folgendermaßen implementiert werden:

```
1   public class DirectedEdge {
2       DirectedEdge(int v, int w, double weight) { /* Erstelle eine
            gerichtete Kante von v zu w mit gegebenem Gewicht */ }
3       double weight() { /* Gib das Gewicht dieser Kante zurück */ }
4       int from() { /* Gib den Ursprungsknoten dieser Kante zurück */ }
5       int to() { /* Gib den Zielknoten dieser Kante zurück */ }
6       String toString() { /* String Darstellung der Kante */ }
7   }
```

7.4 Allgemeine Graphenbegriffe

Unabhängig davon, ob ein Graph gerichtet oder ungerichtet ist, gibt es einige grundlegende Begriffe, die verwendet werden, um bestimmte Aspekte eines Graphen zu beschreiben:

- **Pfad**: Eine Folge von Knoten, die durch Kanten verbunden sind. Dies ist ein grundlegender Begriff, der verwendet wird, um die Verbindung zwischen zwei Knoten in einem Graphen zu beschreiben.
- **Zyklus**: Ein Pfad, dessen erster und letzter Knoten gleich sind. Die Identifizierung von Zyklen ist wichtig für viele Algorithmen und Problemstellungen.
- **Verbunden**: Zwei Knoten sind verbunden, wenn es zwischen ihnen einen Pfad gibt. Dies ist ein grundlegendes Konzept für die Konstruktion und Analyse von Graphen.

7.5 Graphverarbeitungsprobleme

Beim Arbeiten mit Graphen, ob gerichtet oder ungerichtet, gibt es einige gängige Probleme, die oft auftreten:

- **Pfadfindung**: Gibt es zwischen zwei bestimmten Knoten einen Pfad und falls ja, welcher ist der kürzeste?
- **Zyklusdetektion**: Existiert ein Zyklus im Graphen? Speziell interessieren hier Euler-Touren (Zyklen, die jede Kante genau einmal verwenden) und Hamilton-Touren (Zyklen, die jeden Knoten genau einmal verwenden).
- **Konnektivität**: Gibt es eine Möglichkeit, alle Knoten zu verbinden und falls ja, was ist die kosteneffizienteste Methode (Minimale Spannbäume, MSTs)?
- **Bikonnektivität**: Gibt es einen Knoten, dessen Entfernung den Graphen in mehrere, nicht verbundene Teile zerlegt?
- **Planarität**: Kann der Graph so gezeichnet werden, dass keine Kanten sich kreuzen?
- **Graphisomorphismus**: Stellen zwei verschiedene Darstellungen denselben Graphen dar?

Diese Probleme können unterschiedlich komplex sein und einige können sogar unlösbar sein.

7.6 Repräsentation von Graphen

Es gibt verschiedene Möglichkeiten, einen Graphen zu repräsentieren, die je nach Anwendung und Kontext unterschiedliche Vorteile haben können:

7.6.1 Grafische Darstellung

Dies bietet eine visuelle Intuition über die Struktur des Graphen, kann aber irreführend sein, da unterschiedliche Darstellungen denselben Graphen repräsentieren können.

7.6.2 Repräsentation von Knoten

In dieser Vorlesung verwenden wir Integer zwischen 0 und $V - 1$ zur Darstellung von Knoten. In Anwendungen können Namen und Integer-Werte mittels einer Symboltabelle konvertiert werden.

7.6.3 Kantensammlung

Eine weitere Möglichkeit, einen Graphen zu repräsentieren, besteht darin, eine Liste von Kanten zu führen. Dies kann eine verkettete Liste oder ein Array sein.

7.6.4 Adjazenzmatrix

Bei dieser Methode wird ein zweidimensionales Boolean-Array der Größe $V \times V$ verwendet. Für jede Kante `v`–`w` im Graphen werden die Werte `adj[v][w]` (und `adj[w][v]`, falls der Graph ungerichtet ist) auf 'true' (also 1) gesetzt.

7.6.4.1 Implementierung Die Implementierung in Java ist wie folgt:

```
1  public class Graph {
2      private final int V;
3      private final boolean[][] adjMatrix;
4
5      public Graph(int V) {
6          this.V = V;
7          adjMatrix = new boolean[V][V];
8      }
9
10     public void addEdge(int v, int w) {
11         adjMatrix[v][w] = true;
12         adjMatrix[w][v] = true;
13     }
14
15     public Iterable<Integer> adj(int v) {
16         List<Integer> adjV = new ArrayList<Integer>();
17         for (int i = 0; i < V; i++) {
18             if (adjMatrix[v][i]) {
19                 adjV.add(i);
20             }
21         }
22         return adjV;
23     }
24 }
```

Für gerichtete Graphen entferne `adjMatrix[w][v] = true;` in der addEdge-Methode.

7.6.5 Adjazenzliste

Hier wird ein Array von Listen verwendet, das von den Knoten indiziert ist. Jede Liste enthält die Knoten, die an den entsprechenden Knoten angrenzen. Dies ist eine effiziente Repräsentation, insbesondere für "Sparse" (d.h. wenige Kanten) Graphen.

Für gewichtete Digraphen wird neben den Kanten auch noch das Gewicht abgespeichert.

7.6.5.1 Implementierung Für Graphen ohne Gewichte. Bei gerichteten Graphen muss `adj[w].add(v);` entfernt werden.

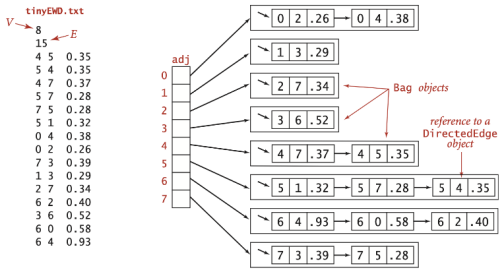


Abbildung 14: Adjazenzliste eines gewichteten Digraphs

```java
public class Graph
{
    private final int V;
    private final Bag<Integer>[] adj;

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w)
    {
        adj[v].add(w);
        adj[w].add(v); // For undirected graph, we add the edge in both directions.
    }

    public Iterable<Integer> adj(int v)
    {
        return adj[v];
    }
}
```

Für gewichtete Digraphen:

```java
public class EdgeWeightedDigraph {
    private final int V;                        // Number of vertices
    private final Bag<DirectedEdge>[] adj;    // Array of bags to store adjacency lists

    public EdgeWeightedDigraph(int V) {
        this.V = V;
        adj = (Bag<DirectedEdge>[]) new Bag[V];    // Creating an array of bags
        for (int v = 0; v < V; v++)
```

```
 9              adj[v] = new Bag<DirectedEdge>();      // Initializing each
                    bag in the array
10      }
11
12      public void addEdge(DirectedEdge e) {
13          int v = e.from();
14          adj[v].add(e);    // Adding a directed edge to the adjacency
                list of vertex v
15      }
16
17      public Iterable<DirectedEdge> adj(int v) {
18          return adj[v];    // Returning the adjacency list of vertex v
19      }
20  }
```

7.7 Tiefensuche (Depth-First Search, DFS)

Die Tiefensuche ist ein Algorithmus zum Durchlaufen und Durchsuchen eines Graphens.

Die Idee ist von der Wurzel (bzw. irgendeinem Knoten im Graphen) rekursiv über alle Knoten zu gehen. Dabei wird jeder bereits besuchte Knoten markiert und nicht erneut besucht. Das bedeutet, startet man DFS bei v, dann besucht der Algorithmus alle w, die an v angrenzen. Als Erstes sucht er alle x, die an das erste w angrenzen, usw. Das bedeutet, wenn w und z an v angrenzen und $v - w$ das kleinere Kantengewicht hat, dann werden erst alle an w angrenzende Knoten besucht.

Mögliche Anwendungen sind:

- Finde alle Knoten, die mit einem gegebenen Ausgangsknoten verbunden sind.
- Finde einen Pfad zwischen zwei Knoten.

7.7.1 Implementierung

Generell ist es vorteilhaft den Graphen von der Graphenverarbeitung zu trennen. Wir erstellen also eine Klasse, die einen Graphen und einen Startpunkt annimmt: `Paths(Graph G, int s)`. Die Klasse hat zwei Funktionen:

- `boolean hasPathTo(int v)`: Gibt `true` zurück, wenn ein Pfad zwischen s und v existiert.
- `Iterable<Integer> pathTo(int v)`: Gibt den Pfad von s zu v zurück, wenn er existiert. Ansonsten gibt die Funktion null zurück.

Zur effektiven Implementierung erstellen wir 2 Arrays für die Tiefensuche:

- `boolean[] marked`: Um besuchte Knoten zu markieren.

- `int[] edgeTo`: Enthält die Pfade durch den Graphen. `edgeTo[w] == v` bedeutet, dass v-w genommen wurde, um w das erste Mal zu besuchen. Dadurch lässt sich also der Pfad von w zu dem Startpunkt finden. Dadurch ist das Array effektiv eine Repräsentation des Graphens als Baum mit dem Startpunkt als Wurzel.

```java
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public DepthFirstSearch(Graph G, int s) {
        // Initialisierung der Klasse
        dfs(G, s); // Finde alle Knoten, die mit s verbunden sind
    }

    private void dfs(Graph G, int v) {
        marked[v] = true; // aktuellen Knoten markieren
        for (int w : G.adj(v)) { // benachbarte Knoten bekommen
            if (!marked[w]) { // wenn nicht bereits markiert
                dfs(G, w); // Rekursion
                edgeTo[w] = v; // Pfad von v zu w speichern
            }
        }
    }
}
```

Bis auf die Klassennamen muss nichts geändert werden für Digraphen.

7.7.2 Analyse

DFS markiert garantiert alle Knoten, die mit s verbunden sind in einer konstanten Zeit. Außerdem wird jeder Knoten nur einmal besucht. Die Laufzeit ist also $O(V + E)$.

7.7.3 Topologisches Sortieren

DFS kann für topologische Sortierung genutzt werden: siehe dieses Video.

7.8 Breitensuche (Breadth-First Search, BFS)

Die Breitensuche ist ein Algorithmus zum Durchlaufen und Durchsuchen eines Graphens.

Die grundlegende Idee ist, von der Wurzel (oder einem beliebigen Knoten im Graphen) auszugehen und alle benachbarten Knoten zu besuchen, bevor man zu den nächsten Ebenen übergeht (Ebene i heißt Knoten, dessen Pfad Länge i zu der Wurzel hat). Jeder besuchte Knoten wird markiert und nicht

erneut besucht. Das bedeutet, wenn man BFS bei v startet, dann besucht der Algorithmus zuerst alle w, die an v angrenzen, bevor er weiter zu den Knoten geht, die an w angrenzen.

Mögliche Anwendungen sind:

- Finden aller Knoten, die mit einem gegebenen Ausgangsknoten verbunden sind.
- Finden des kürzesten Pfades zwischen zwei Knoten.

7.8.1 Implementierung

Wie bei DFS ist es auch bei BFS vorteilhaft, den Graphen von der Graphenverarbeitung zu trennen. Wir erstellen also eine Klasse, die einen Graphen und einen Startpunkt annimmt: `Paths(Graph G, int s)`. Die Klasse hat zwei Funktionen:

- `boolean hasPathTo(int v)`: Gibt `true` zurück, wenn ein Pfad zwischen s und v existiert.
- `Iterable<Integer> pathTo(int v)`: Gibt den Pfad von s zu v zurück, wenn er existiert. Ansonsten gibt die Funktion null zurück.

Für die Breitensuche erstellen wir zwei Arrays:

- `boolean[] marked`: Um besuchte Knoten zu markieren.
- `int[] edgeTo`: Enthält die Pfade durch den Graphen. `edgeTo[w] == v` bedeutet, dass v-w genommen wurde, um w das erste Mal zu besuchen. Dadurch lässt sich also der Pfad von w zu dem Startpunkt finden.

```java
public class BreadthFirstPaths {
    private boolean[] marked;
    private boolean[] edgeTo;
    private final int s;

    // ...

    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    q.enqueue(w);
                    marked[w] = true;
                    edgeTo[w] = v;
                }
            }
```

```
22              }
23          }
24      }
```

Bis auf die Klassennamen muss nichts geändert werden für Digraphen.

7.8.2 Analyse

BFS garantiert, dass alle Knoten, die mit s verbunden sind, in einer konstanten Zeit markiert werden. Wie DFS besucht auch BFS jeden Knoten nur einmal. Die Laufzeit ist also ebenfalls $O(V + E)$.

Für ungerichtete Graphen gilt außerdem: Im Unterschied zu DFS liefert BFS jedoch immer den kürzesten Pfad zwischen dem Startknoten und jedem anderen erreichbaren Knoten.

7.8.3 Finden des kürzesten Pfads

BFS kann den kürzesten Pfad mit mehreren Startpunkten finden. Dafür muss am Anfang von der BFS Funktion alle Startpunkte in die Queue hinzugefügt werden.

7.9 Zusammenhangskomponenten

Eine Zusammenhangskomponente ist eine maximale Menge an verbundenen Knoten. Es ist also ein Untergraphen eines ungerichteten Graphen, in dem jeder Knoten mit jedem anderen Knoten durch einen Pfad verbunden ist (Äquivalenzrelation, siehe Union Find). In anderen Worten, es gibt keine zwei Knoten in einer Zusammenhangskomponente zwischen denen kein Pfad existiert. Darüber hinaus hat eine Zusammenhangskomponente keine Verbindung zu anderen Knoten außerhalb der Komponente.

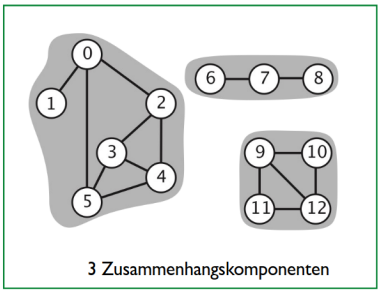
Um in konstanter Zeit herauszufinden, ob v mit w verbunden ist, speichern wir alle Zusammenhangskomponenten eines Graphens ab.

7.9.1 Algorithmus

Die Zusammenhangskomponenten eines Graphen können mithilfe von Graphentraversierungsalgorithmen wie Tiefensuche (DFS) oder Breitensuche (BFS) gefunden werden.

Hier ist ein Beispielalgorithmus, der DFS verwendet, um die Zusammenhangskomponenten eines Graphen zu bestimmen:

1. Markiere alle Knoten als unbesucht.



Zusammenhangskomponenten ID (leicht mit DFS zu berechnen)

	0	1	2	3	4	5	6	7	8	9	10	11	12
cc[]	0	0	0	0	0	0	1	1	1	2	2	2	2

Abbildung 15: Beispiel von Zusammenhangskomponenten eines Graphens

2. Wähle einen unbesuchten Knoten und starte eine Tiefensuche von diesem Knoten aus. Alle Knoten, die von diesem Knoten aus erreichbar sind, bilden eine Zusammenhangskomponente.
3. Wiederhole Schritt 2 mit einem anderen unbesuchten Knoten, falls vorhanden, um eine weitere Zusammenhangskomponente zu finden.
4. Wiederhole dies, bis alle Knoten besucht wurden.

Die Anzahl der Durchläufe der Tiefensuche gibt die Anzahl der Zusammenhangskomponenten an. Die spezifischen Knoten, die während eines Durchlaufs besucht wurden, bilden eine Zusammenhangskomponente.

7.9.2 Implantation

Wir erstellen eine Klasse `CC(Graph G)` (Zusammenhangskomponenten = Connected Components = CC), mit den folgenden Funktionen:

- **`boolean connected(int v, int w)`**: Gibt true zurück, falls v und w verbunden sind
- **`int count()`**: Gibt die Anzahl an Zusammenhangskomponenten zurück
- **`int id(int v)`**: Gibt die ID der Zusammenhangskomponte, in der v ist.

```
1  public class CC
2  {
3      private boolean marked[];  // Array zum Markieren der besuchten
           Knoten
4      private int[] id;  // Array zur Speicherung der ID jeder
           Zusammenhangskomponente
5      private int count;  // Zählt die Anzahl der
           Zusammenhangskomponenten
6
7      public CC(Graph G)
8      {
```

```
9           marked = new boolean[G.V()];  // Initialisiert das Array mit
                der Anzahl der Knoten im Graphen
10          id = new int[G.V()];  // Dasselbe für die id-Array
11          for (int v = 0; v < G.V(); v++)  // Durchläuft alle Knoten im
                Graphen
12          {
13              if (!marked[v])  // Wenn der Knoten noch nicht markiert
                    wurde
14              {
15                  dfs(G, v);  // Führt eine Tiefensuche auf diesem Knoten
                        aus
16                  count++;  // Erhöht die Anzahl der
                        Zusammenhangskomponenten
17              }
18          }
19      }
20
21      public int count()
22      {
23          return count;
24      }
25
26      public int id(int v)
27      {
28          return id[v];
29      }
30
31      private void dfs(Graph G, int v)
32      {
33          marked[v] = true;  // Markiert den Knoten als besucht
34          id[v] = count;  // Weist dem Knoten die aktuelle count-ID zu
35          for (int w : G.adj(v))  // Durchläuft alle Nachbarknoten von v
36              if (!marked[w])  // Wenn der Nachbarknoten noch nicht
                    markiert wurde
37                  dfs(G, w);  // Führt eine Tiefensuche auf diesem Knoten
                        aus
38      }
39  }
```

7.9.3 Analyse

Wird abgefragt, ob 2 Knoten verbunden sind, muss lediglich die ID der beiden identisch sein. Die Laufzeit ist also linear.

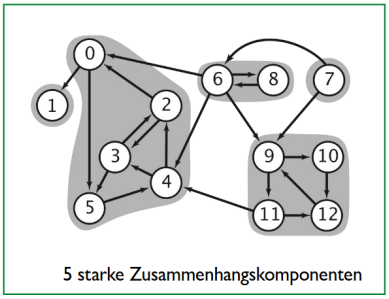
Implementierung in Java:

```
1  public int connected(int v, int w)
2  { return cc[v] == cc[w]; }
```

7.9.4 Starke Zusammenhangskomponente

2 Knoten v und w sind stark zusammenhängend, wenn es einen direkten Pfad von v nach w und einen direkten Pfad von w nach v gibt. Starker Zusammenhang ist also eine Äquivalenzrelation.

Eine starke Zusammenhangskomponente ist eine maximale Untermenge an stark zusammenhängenden Knoten eines Graphens.



starke Zusammenhangskomponenten ID (wie berechenbar?)

	0	1	2	3	4	5	6	7	8	9	10	11	12
scc[]	1	0	1	1	1	1	3	4	3	2	2	2	2

7.10 Minimum Spanning Trees (MSTs)

Ein Spanning Tree von einem Graphen G ist ein Subgraph T, der

- verbunden ist,
- azyklisch ist und
- alle Knoten beinhaltet.

Ein Minimum Spanning Tree ist ein Spanning Tree eines gewichteten Digraphs, dessen Gewicht (also Summe der Gewichte aller Kanten) kleinstmöglich ist. Zur Vereinfachung nehmen wir an, dass der Graph verbunden ist und alle Kantengewichte unterschiedlich sind. Daraus folgt, dass der MST existiert und eindeutig ist.

Schnitteigenschaft: Die Schnitteigenschaft zeigt, welche Kanten Teil eines MSTs sein müssen. Ein Schnitt in einem Graphen ist eine Partition seiner Knoten in zwei Mengen (Beispiel: Rot-markierten Kanten in Abbildung 16). Eine Querkante verbindet einen Knoten in einer Menge mit einem Knoten in einer anderen Menge. Die Querkante mit dem kleinsten Gewicht muss Teil des MSTs sein. (Siehe Abbildung 16.)

Im folgenden Teil werden wir zwei Greedy Algorithmen betrachten: Kruskal's und Prim's Algorithmus. Greedy Algorithmen sind Algorithmen, die an jeder Stelle die lokal optimale Wahl treffen in der Hoffnung, ein globales Optimum zu finden (heißt, das globale Optimum wird nicht garantiert gefunden).

Generell funktionieren Greedy Algorithmen zur Berechnung von MSTs folgendermaßen:

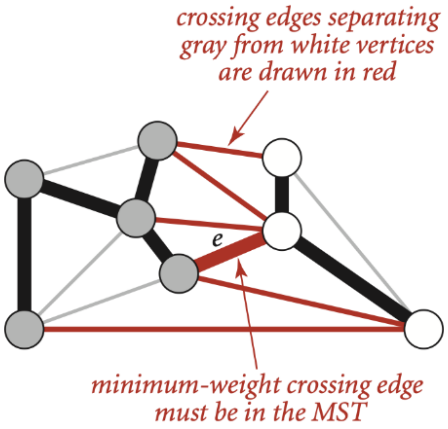


Abbildung 16: Schnitteigenschaft Darstellung

1. Starte mit allen Kanten grau gefärbt.
2. Finde einen Schnitt mit Querkanten, die nicht schwarz sind und färbe die Kante mit geringstem Gewicht schwarz.
3. Setze fort bis $V - 1$ Kanten Schwarz gefärbt sind.

Für ein Beispiel siehe Abbildung 17.

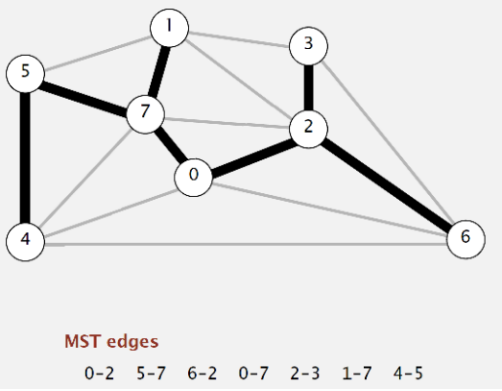


Abbildung 17: Beispiel eines Greedy MST

7.10.1 Kruskal's Algorithmus

Kruskal's Algorithmus sortiert zuerst alle Kanten nach Gewicht (aufsteigend). Anschließend wird immer die kleinste Kante hinzugefügt, die keinen Zyklus bildet, bis $V - 1$ Kanten hinzugefügt wurden.

Um zu überprüfen, ob das Hinzufügen einer Kante einen Zyklus erzeugt, nutzen wir die Union-Find Datenstruktur. Für jede Kante des MST werden die beiden Knoten der Kante vereinigt. Damit eine

Kante Teil des MST ist, müssen die beiden Knoten der Kante nicht Teil der selben Menge innerhalb der Union-Find Datenstruktur sein.

7.10.1.1 Implementierung

Die Implementierung in Java ist wie folgt:

```java
public class KruskalMST {
    private Queue<Edge> mst = new Queue<Edge>(); // Queue to store the MST edges

    public KruskalMST(EdgeWeightedGraph G) {
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges()); // Priority queue to store the graph edges
        UF uf = new UF(G.V()); // Union-Find data structure to track connected components

        while (!pq.isEmpty() && mst.size() < G.V() - 1) {
            Edge e = pq.delMin(); // Get the minimum-weight edge from the priority queue
            int v = e.either(), w = e.other(v); // Get the vertices of the edge

            if (!uf.connected(v, w)) {
                uf.union(v, w); // Union the components of v and w
                mst.enqueue(e); // Add the edge to the MST
            }
        }
    }

    public Iterable<Edge> edges() {
        return mst; // Return the MST edges
    }
}
```

7.10.1.2 Laufzeit

Kruskal's Algorithmus berechnet einen MST in Zeit proportional zu $O(E \log E)$. Die Laufzeiten für die verschiedenen Operationen sind:

Operation	Häufigkeit	Zeit pro Operation
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V$ (amorized WQUPC)
connected	E	$\log^* V$ (amorized WQUPC)

(Wiederholung: $\log^* V \leq 5$)

7.10.2 Prim's Algorithmus (lazy)

Prim's Algorithmus (in der lazy Variante) beginnt mit einem beliebigen Knoten und fügt immer die kleinste Kante hinzu, die zu einem bereits verbundenen Knoten führt, aber keinen Zyklus bildet. Das geschieht, bis $V - 1$ Kanten hinzugefügt wurden.

Um die kleinste Kante zu bestimmen, die zu einem bereits verbundenen Knoten führt, verwenden wir eine Priority Queue. Im "lazy" Prim's Algorithmus werden Kanten zur Priority Queue hinzugefügt, selbst wenn sie später nicht benötigt werden, daher die Bezeichnung "lazy". Wenn durch `delete-min` die nächte Kante aus der Priority Queue genommen wird, dann prüft der Algorithmus erst, ob die Knoten der Kante nicht Teil des MSTs sind.

7.10.2.1 Implementierung

Die Implementierung in Java ist wie folgt:

```java
public class LazyPrimMST {
    private boolean[] marked; // Marked vertices
    private Queue<Edge> mst; // Minimum spanning tree
    private MinPQ<Edge> pq; // Priority queue for edges

    public LazyPrimMST(EdgeWeightedGraph G) {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];

        visit(G, 0); // Start from vertex 0

        while (!pq.isEmpty()) {
            Edge e = pq.delMin(); // Get the minimum-weight edge from
                the priority queue

            int v = e.either(), w = e.other(v); // Get the vertices of
                the edge
            if (marked[v] && marked[w]) continue; // Skip if redundant
                edge

            mst.enqueue(e); // Add the edge to the MST

            if (!marked[v]) visit(G, v); // Visit vertex v
            if (!marked[w]) visit(G, w); // Visit vertex w
        }
    }

    private void visit(EdgeWeightedGraph G, int v) {
        // Mark vertex v and add to pq all edges from v to unmarked
            vertices
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
```

```
31      }
32
33      public Iterable<Edge> edges() {
34          return mst; // Return the MST edges
35      }
36  }
```

7.10.2.2 Laufzeit Prim's Algorithmus berechnet einen MST in Zeit proportional zu $O(E \log E)$. Die Laufzeiten für die verschiedenen Operationen sind:

Operation	Häufigkeit	Zeit pro Operation
insert	E	$\log E$
del-min	E	$\log E$

7.10.3 Prim's Algorithmus (eager)

Im Gegensatz zur "lazy" Variante führt Prim's "eager" Algorithmus zusätzliche Arbeit aus, um sicherzustellen, dass nicht benötigte Kanten nicht in der Priority Queue bleiben. Genauer gesagt wollen wir Kanten finden, die mit genau einem Endpunkt des MSTs verbunden sind und minimales Gewicht haben.

Dafür setzen wir eine indexierte PQ ein, welche aber Knoten hält, die über eine Kante mit dem MST verbunden sind. Die Priorität der Knoten ist gegeben durch das Gewicht der Kante, die die Knoten mit dem MST verbinden.

Wenn der nächste Knoten v aus der PQ geholt wird, dann geschieht Folgendes:

- Füge Kante $e = v - w$ zu dem MST hinzu (w ist Teil des MSTs)
- Finde alle Kanten $e = v - x$ und
    a. ignoriere, falls x in MST,
    b. füge x zu PQ hinzu oder
    c. Reduziere die Priorität von x, falls $v - x$ die kürzeste Kante ist, um x mit dem MST zu verbinden.

7.10.3.1 Laufzeit

PQ Implementierung	insert	delete-min	decrease-key	total
Array	1	V	1	V^2
Binary Heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way Heap	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{EN} V$
Fibonacci Heap	1^*	$\log V^*$	1^*	$E + V \log V$

$*$: amortisiert

Kernaussage:

- Array Implementierung optimal für dichte Graphen
- Binary Heap viel schneller für wenig dichte Graphen
- 4-way Heap lohnend für Performance kritische Situation
- Fibonacci Heap optimal in der Theorie; Implementierung lohnt sich aber nicht

7.11 Shortest Path (Kürzester Pfad)

Der kürzeste Pfad ist ein fundamentales Konzept in der Theorie der Graphen und findet Anwendung in vielen Bereichen, von Netzwerk-Routing bis hin zu Transportlogistik und Navigation. Es handelt sich dabei um die Suche nach dem Pfad mit der geringsten Gesamtgewichtung zwischen zwei Knoten in einem gewichteten Graphen.

Für einen optimalen SPT (Shortest Path Tree) eines gewichteten Digraphs G von dem Knoten s aus, muss gelten:

- `distTo[s]` $= 0$
- Für jeden Knoten v, ist `distTo[v]` die Länge eines Pfades von s nach v
- Für jede Kante $e = v \to w$, muss gelten `distTo[w]` $\leq$ `distTo[v] + e.weight()`

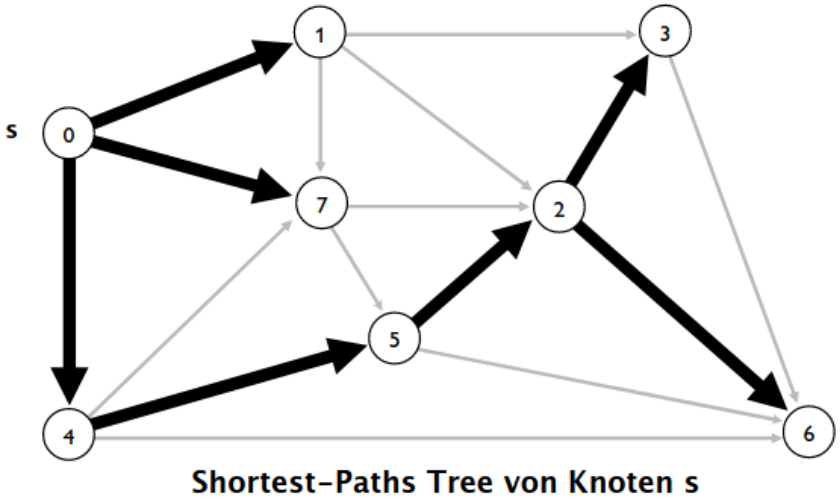
Die Berechnung eines optimalen SPTs funktioniert wie folgt:

1. Initialisiere `distTo[s]`=`0` und `distTo[v]`=∞ für alle anderen Knoten `v`.
2. Wiederhole bis Optimilitätsbedingung erfüllt ist: Relaxiere eine Kante.

In den folgenden Abschnitten werden wir uns mit 2 Algorithmen befassen, die bestimmen welche Kanten relaxiert werden.

Bei gewichteten DAGs (Directed Acyclic Graphs = gerichtete azyklische Graphen) gibt es eine leichtere Methode, um ein optimalen SPT zu bestimmen (siehe Beispiel: Abbildung 18):

- Betrachte alle Knoten in topologischer Reihenfolge (z.B. von Knoten 0 bis x)
- Relaxiere alle Kanten, die von den Knoten weg zeigen



0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Abbildung 18: Beispiel SPT eines DAGs

Topologische Sortieralgorithmen funktionieren auch mit **negativen Kantengewichten**. Dies können wir nutzen, um den längsten Pfad eines DAGs zu finden:

1. Negiere alle Gewichte
2. Finde den kürzesten Pfad
3. Negiere Gewichte im Ergebnis

7.11.1 Implementierung

Hier ist eine Java-Implementierung für den kürzesten Pfad, die auf dem Konzept der kantengewichteten Graphen (EdgeWeightedDigraph) basiert.

```java
public class ShortestPath {
    private EdgeWeightedDigraph G;
    private int s;
    private double[] distTo;
    private DirectedEdge[] edgeTo;

    public ShortestPath(EdgeWeightedDigraph G, int s) {
        this.G = G;
        this.s = s;
        // Initialize distTo and edgeTo arrays
    }

    public double distTo(int v) {
```

```
14          return distTo[v];
15      }
16
17      public Iterable<DirectedEdge> pathTo(int v) {
18          Stack<DirectedEdge> path = new Stack<DirectedEdge>();
19          for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()
                ]) {
20              path.push(e);
21          }
22          return path;
23      }
24
25      public boolean hasPathTo(int v) {
26          // Check if v has a path to s
27      }
28
29      private void relax(DirectedEdge e)
30      {
31          int v = e.from(), w = e.to();
32          if (distTo[w] > distTo[v] + e.weight())
33          {
34              distTo[w] = distTo[v] + e.weight();
35              edgeTo[w] = e;
36          }
37      }
38  }
```

Die Klasse `ShortestPath` enthält folgende Datenfelder und Methoden:

- `G`: Ein kanten-gewichteter Digraph.
- `s`: Der Startknoten.
- `distTo`: Ein Array, das die kürzesten Entfernungen zu allen Knoten vom Startknoten aus speichert.
- `edgeTo`: Ein Array, das den kürzesten Pfad zu jedem Knoten vom Startknoten aus speichert.

Die Klasse enthält folgende Methoden:

- `ShortestPath(EdgeWeightedDigraph G, int s)`: Der Konstruktor der Klasse, der den Graphen `G` und den Startknoten `s` initialisiert. Die Arrays `distTo` und `edgeTo` werden ebenfalls initialisiert.
- `distTo(int v)`: Gibt die kürzeste Entfernung vom Startknoten `s` zum Knoten `v` zurück.
- `pathTo(int v)`: Gibt den kürzesten Pfad vom Startknoten `s` zum Knoten `v` zurück. Dies wird erreicht, indem der Pfad rückwärts durch das Array `edgeTo` verfolgt wird.
- `hasPathTo(int v)`: Überprüft, ob es einen Pfad vom Startknoten `s` zum Knoten `v` gibt.
- `relax(DirectedEdge e)`: Entspannt die Kanten, indem sie das Gewicht der Kante und das aktuelle distTo des Zielknotens vergleicht. Wenn der neue Pfad kürzer ist, wird er aktualisiert.

7.11.2 Dijkstra's Algorithmus

Dijkstra's Algorithmus ist einer der bekanntesten Algorithmen zur Lösung des kürzesten Pfadproblems in einem gewichteten Graphen, in dem alle Kanten positive Gewichte haben. Der Algorithmus wurde 1959 vom niederländischen Computerwissenschaftler Edsger W. Dijkstra entwickelt.

Der Algorithmus funktioniert wie folgt:

1. Setze die kürzeste bekannte Distanz zu jedem Knoten auf unendlich, mit Ausnahme des Startknotens, dessen kürzeste Distanz auf 0 gesetzt wird.
2. Setze den Startknoten als aktuellen Knoten.
3. Für den aktuellen Knoten betrachte alle seine ungeprüften Nachbarn und berechne ihre potenziellen Distanzen durch den aktuellen Knoten. Vergleiche die neu berechnete potenzielle Distanz mit der aktuellen zugewiesenen und speichere die kleinste Distanz.
4. Sobald alle Nachbarn des aktuellen Knotens geprüft wurden, markiere den Knoten als geprüft. Ein geprüfter Knoten wird nicht erneut geprüft.
5. Wenn der Zielknoten geprüft wurde, stoppt der Algorithmus und der Pfad kann bestimmt werden.
6. Wenn nicht, wähle den ungeprüften Knoten, der die geringste Distanz hat, als den nächsten Knoten und wiederhole den Algorithmus ab Schritt 3.

Für ein Beispiel, siehe Abbildung **??**

7.11.2.1 Implementierung

Die Implementierung in Java ist wie folgt:

```java
public class DijkstraSP {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
```

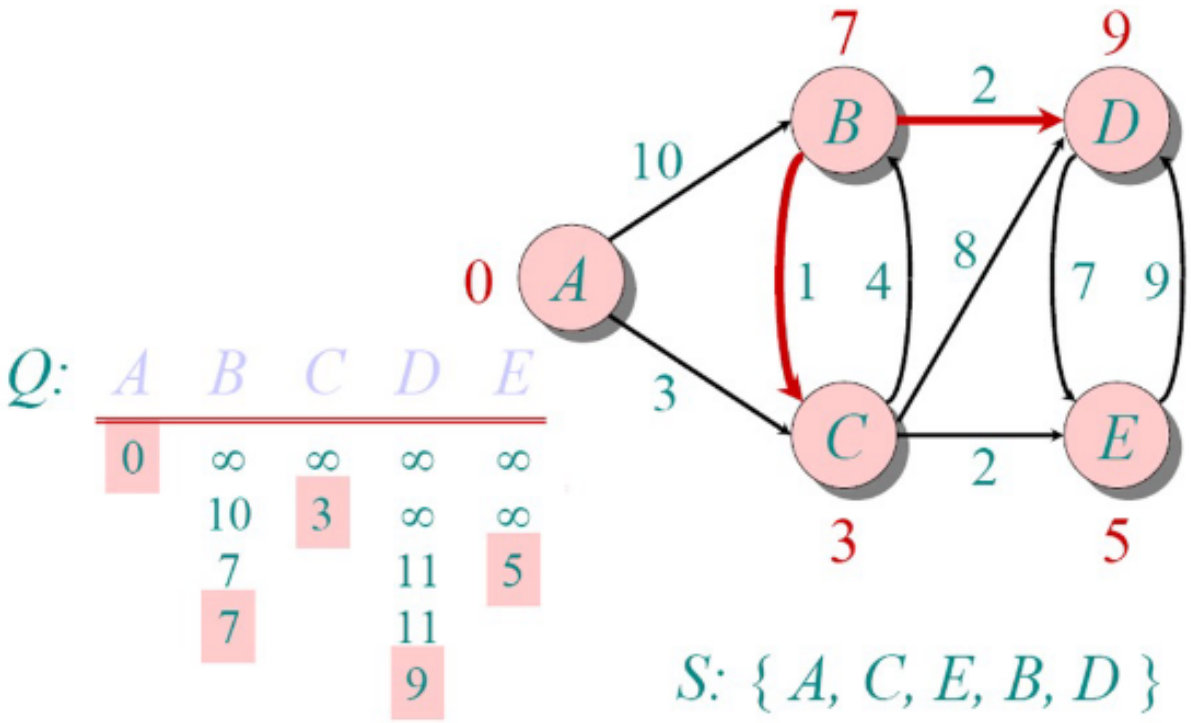


Abbildung 19: Dijkstra Algorithmus Beispiel

```
22
23      private void relax(DirectedEdge e) {
24          int v = e.from(), w = e.to();
25          if (distTo[w] > distTo[v] + e.weight()) {
26              distTo[w] = distTo[v] + e.weight();
27              edgeTo[w] = e;
28              if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
29              else                pq.insert(w, distTo[w]);
30          }
31      }
32
33      public double distTo(int v) {
34          return distTo[v];
35      }
36
37      public boolean hasPathTo(int v) {
38          return distTo[v] < Double.POSITIVE_INFINITY;
39      }
40
41      public Iterable<DirectedEdge> pathTo(int v) {
42          if (!hasPathTo(v)) return null;
43          Stack<DirectedEdge> path = new Stack<DirectedEdge>();
44          for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()
                ])
```

```
45                path.push(e);
46            return path;
47        }
48    }
```

7.11.2.2 Laufzeit

PQ Implementierung	insert	delete-min	decrease-key	total
Array	1	V	1	V^2
Binary Heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way Heap	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{EN} V$
Fibonacci Heap	1^*	$\log V^*$	1^*	$E + V \log V$

$*$: amortisiert

Kernaussage:

- Array Implementierung optimal für dichte Graphen
- Binary Heap viel schneller für wenig dichte Graphen
- 4-way Heap lohnend für Performance kritische Situation
- Fibonacci Heap optimal in der Theorie; Implementierung lohnt sich aber nicht

7.11.3 Bellman-Ford Algorithmus

Dijkstra's Algorithmus funktioniert nicht mit negativen Gewichten. Es ist außerdem nicht möglich alle Gewichte um x zu erhöhen, um negative Gewichte zu entfernen. Es muss also ein anderer Algorithmus her.

Der Bellman-Ford Algorithmus funktioniert wie folgt:

1. Initialisiere `distTo[s]=0` und `distTo[v]=`∞ für alle anderen Knoten.
2. Wiederhole $V - 1$ mal: Relaxiere jede Kante.

Der Code sieht also so aus:

```
1  for (int i = 0; i < G.V() - 1; i++)
2      for (int v = 0; v < G.V(); v++)
3          for (DirectedEdge e : G.adj(v))
4              relax(e);
```

Der Algorithmus hat eine Laufzeit proportional zu $O(E \cdot V)$.

Eine Verbesserung ist eine Queue zu erstellen, die alle Knoten beinhaltet, dessen `distTo` sich verändert hat. Der Worst-Case ändert sich dadurch zwar nicht, aber in der Praxis ist die Performance deutlich besser.

Bellman-Ford ist zwar in der Lage mit negativen Gewichten zu rechnen, aber nicht mit negativen Zyklen (ein Zyklus, dessen Summe der Gewichte der Kanten negativ ist). Der Algorithmus kann aber genutzt werden, um negative Zyklen zu erkennen.

7.11.3.1 Implementierung

Die Implementierung in Java ist wie folgt:

```java
public class BellmanFordSP {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private boolean[] onQ;
    private Queue<Integer> queue;

    public BellmanFordSP(EdgeWeightedDigraph G, int s) {
        int V = G.V();
        distTo = new double[V];
        edgeTo = new DirectedEdge[V];
        onQ = new boolean[V];
        queue = new LinkedList<>();

        for (int v = 0; v < V; v++)
            distTo[v] = Double.POSITIVE_INFINITY;

        distTo[s] = 0.0;
        queue.offer(s);
        onQ[s] = true;

        while (!queue.isEmpty()) {
            int v = queue.poll();
            onQ[v] = false;

            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    private void relax(DirectedEdge e) {
        int v = e.from();
        int w = e.to();

        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;

```

```
38                 if (!onQ[w]) {
39                     queue.offer(w);
40                     onQ[w] = true;
41                 }
42             }
43         }
44     }
```

7.11.4 Zusammenfassung der Algorithmen

Algorithmus	Einschränkung	Typischer Fall	Worst-Case	Extraplatz
Topologische Sortierung	keine gerichteten Zyklen	E + V	E + V	V
Dijkstra (Binary Heap)	keine negativen Gewichte	E log V	E log V	V
Bellman-Ford	keine negativen Zyklen	E V	E V	V
Bellman-Ford (basieren auf Queue)		E + V	E V	V

Abbildung 20: Kosten der Algorithmen zur Bestimmung des SP

7.12 Herausforderungen

7.12.1 Ist ein Graph bipartit?

Problem: Ein Graph ist bipartit, wenn die Knoten in zwei disjunkte Mengen aufgeteilt werden können, so dass jede Kante des Graphen Knoten aus unterschiedlichen Mengen verbindet.

Schwierigkeit: Mittel. Das Problem kann in linearer Zeit (O(V+E)) gelöst werden.

Lösungsalgorithmus: Eine Methode zur Überprüfung, ob ein Graph bipartit ist oder nicht, ist die Verwendung einer modifizierten Breitensuche (Breadth-First Search, BFS).

1. Wähle einen zufälligen Knoten aus und weise ihm eine Farbe zu (z.B. “rot”).
2. Färbe alle seine Nachbarn mit der anderen Farbe (z.B. “blau”).
3. Wiederhole den Prozess für alle unbesuchten Knoten und wechsle dabei immer die Farben.
4. Wenn du während des Färbungsprozesses auf einen Knoten triffst, der bereits die gleiche Farbe wie der aktuelle Knoten hat, dann ist der Graph nicht bipartit.

7.12.2 Finde einen Zyklus

Problem: Ein Zyklus in einem Graphen ist ein Pfad, der mit demselben Knoten beginnt und endet.

Schwierigkeit: Mittel. Ähnlich wie das vorherige Problem kann auch dieses in linearer Zeit (O(V+E)) gelöst werden.

Lösungsalgorithmus: Ein Zyklus kann durch eine Tiefensuche (Depth-First Search, DFS) gefunden werden.

1. Starte die DFS von jedem noch nicht besuchten Knoten.
2. Während der DFS, wenn du einen bereits besuchten Knoten erreichst, der nicht der direkte Vorgänger ist, hast du einen Zyklus gefunden.
3. Merke dir den Pfad, der zum Zyklus führt, indem du die Vorgänger der Knoten speicherst.

7.12.3 Finde einen Zyklus, der jede Kante genau einmal verwendet (Euler Tour)

Problem: Ein Euler-Zyklus in einem Graphen ist ein Pfad, der jede Kante genau einmal verwendet und zum Ausgangspunkt zurückkehrt. Ein Graph enthält einen Euler-Zyklus, wenn er zusammenhängend ist und alle seine Knoten einen geraden Grad haben.

Schwierigkeit: Mittel. Das Problem kann in linearer Zeit in Bezug auf die Anzahl der Kanten (O(E)) gelöst werden.

Lösungsalgorithmus: Eine Methode zum Finden eines Euler-Zyklus ist der Fleury-Algorithmus.

1. Wähle einen zufälligen Knoten als Startpunkt.
2. Folge den Kanten und entferne sie, wenn du sie durchläufst.
3. Wähle immer eine Kante, die keine Brücke ist (es sei denn, es gibt keine andere Wahl). Eine Brücke ist eine Kante, deren Entfernung den Graphen in zwei nicht verbundene Teile teilt (= eine Kante ist eine Brücke, wenn sie nicht Teil eines Zyklus ist).
4. Wiederhole diesen Prozess, bis alle Kanten besucht wurden.

7.12.4 Finde einen Zyklus, der jeden Knoten genau einmal besucht (Hamilton Tour)

Problem: Ein Hamilton-Zyklus in einem Graphen ist ein Pfad, der jeden Knoten genau einmal besucht und zum Ausgangspunkt zurückkehrt. Im Gegensatz zum Euler-Zyklus gibt es für das Hamilton-Zyklus-Problem keinen effizienten Algorithmus, da es zu den NP-vollständigen Problemen gehört.

Schwierigkeit: Hoch. Das Problem ist NP-vollständig.

Lösungsalgorithmus: Es gibt verschiedene Algorithmen zur Lösung dieses Problems, wie den Backtracking-Algorithmus.

1. Wähle einen beliebigen Knoten als Startpunkt.
2. Besuche einen benachbarten Knoten, wenn dieser noch nicht besucht wurde.
3. Wiederhole diesen Prozess, bis entweder alle Knoten besucht wurden und du zum Ausgangspunkt zurückkehren kannst (in diesem Fall hast du einen Hamilton-Zyklus gefunden) oder es keinen benachbarten Knoten mehr gibt, den du besuchen kannst (in diesem Fall geh zurück und versuche eine andere Route).

7.12.5 Überprüfe, ob bei 2 Graphen ein Isomorphismus vorliegt

Problem: Zwei Graphen sind isomorph, wenn sie die gleiche Struktur haben, d.h. es gibt eine eindeutige Zuordnung (Bijektion) der Knoten des einen Graphen zu den Knoten des anderen Graphen, so dass die Verbindungen zwischen den Knoten erhalten bleiben.

Schwierigkeit: Hoch. Das Graph-Isomorphie-Problem ist ein bekanntes offenes Problem in der Informatik und es ist bisher nicht bekannt, ob es in Polynomialzeit lösbar ist oder nicht.

Lösungsalgorithmus: Es gibt verschiedene Algorithmen zur Lösung dieses Problems, aber keiner davon ist effizient für alle Graphen. Ein gängiger Algorithmus ist der VF2 (Vento-Foggia) Algorithmus, der auf einer heuristischen Suche basiert.

7.12.6 Ordne einen Graphen in der Ebene so an, dass sich keine Kanten kreuzen

Problem:: Ein planarer Graph ist ein Graph, der so auf der Ebene gezeichnet werden kann, dass sich keine Kanten kreuzen. Das Problem besteht darin, zu prüfen, ob ein gegebener Graph planar ist und gegebenenfalls eine solche Anordnung zu finden.

Schwierigkeit: Mittel bis hoch, abhängig von der Komplexität des Graphen. Die Planaritätsprüfung kann in linearer Zeit (O(V+E)) durchgeführt werden, aber die Erstellung einer planaren Einbettung kann schwieriger sein.