Laufzeitanalyse

Programmieren und Software-Engineering
Theorie

22. Februar 2023

Sortieren: Insertion-Sort

Quelle: [?]

Algorithm 1: Insertion Sort

```
1  Function InsertionSort(array a[])
       Result: sorted array a
2      i = 1;
3      while i < a.length do
4          j = i;
5          while j > 0 ∧ a[j − 1] > a[j] do
6              swap(a, j, j − 1);
7              j = j − 1;
8          i = i + 1;
```

- Algorithmus funktioniert wie Karten-Sortieren in der Hand

Sortieren: Selection-Sort

Algorithm 2: Selection Sort

```
1  Function SelectionSort(array a[])
       Result: sorted array a
2      for i = 0; i < a.length − 1; i++ do
3          jMin = i;
4          for j = i + 1; j < a.length; j++ do
5              if a[j] < a[jMin] then
6                  jMin = j;
7          if jMin ≠ i then
8              swap(a, i, jMin);
```

Sortieren: Selection-Sort

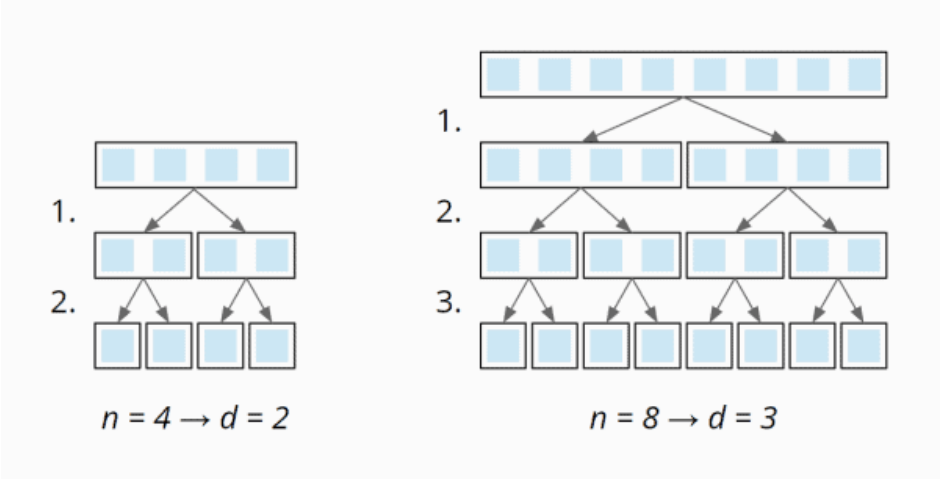
Algorithm 3: Selection Sort

```
1  Function SelectionSort(array a[])
       Result: sorted array a
2      for i = 0; i < a.length − 1; i++ do
3          jMin = i;
4          for j = i + 1; j < a.length; j++ do
5              if a[j] < a[jMin] then
6                  jMin = j;
7          if jMin ≠ i then
8              swap(a, i, jMin);
```

Analyse:

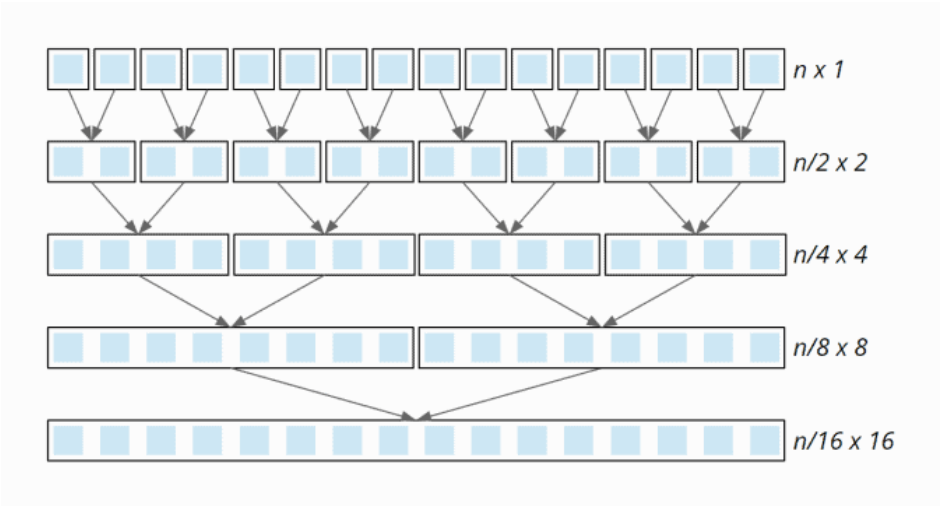
- Sei $n = a.length$
- Die Schleife in Zeile 2 wird $n-1$ mal durchlaufen
- Durchläufe der inneren Schleife: $(n-1), (n-2), \ldots 1$
- Somit $(n-1) + (n-2) + \ldots 1 = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n-1)$

Analyse von Mergesort



- Insgesamt gibt es $\mathrm{ld}n$ Ebenen der Unterteilung.
- Bei der doppelten Anzahl an Elementen gibt es nur eine zusätzliche Ebene.
- ld steht für den Logarithmus dualis, also jenen zur Basis 2 (da ja immer in zwei Teile geteilt wird).

Analyse von Mergesort



- Ebenso gibt es $\mathrm{ld}\, n$ Merge-Ebenen.
- In jeder Ebene (bei Unterteilung und Merge) werden n Elemente betrachtet.
- Somit ergibt sich eine Gesamtlaufzeit von $O(n \log n)$.

Analyse von Quicksort

- Analyse von Quicksort insgesamt schwieriger (als bei Mergesort)
- Im **Worst Case** $O(n^2)$
- Dieser wird jedoch nur sehr selten erreicht.
- Im **Durschnittsfall** hat Quicksort eine Laufzeit von $O(n \log n)$
- Somit: gleiche theoretische Laufzeit wie Mergesort
- In der Praxis jedoch schneller als Mergesort!
- Sehr häufig verwendetes Sortierverfahren (z.B. in Bibliotheken)!

Vergleich Sortier-Algorithmen

Vergleich der Laufzeiten von Sortier-Algorithmen im **Worst-Case**. Worst Case bedeutet, dass die Eingabedaten derart ungünstig sortiert sind, dass der Algorithmus die maximale Anzahl an Schritten ausführt. Der Average-Case betrachtet das *durchschnittliche* Verhalten.

Worst-Case Analyse einfacher Sortieralgorithmen:

Algorithmus	**Vergleiche**	**Swaps**
Bubble-Sort	$O(n^2)$	$O(n^2)$
Insertion-Sort	$O(n^2)$	$O(n^2)$
Selection-Sort	$O(n^2)$	$O(n)$

Analyse weiterer Algorithmen:

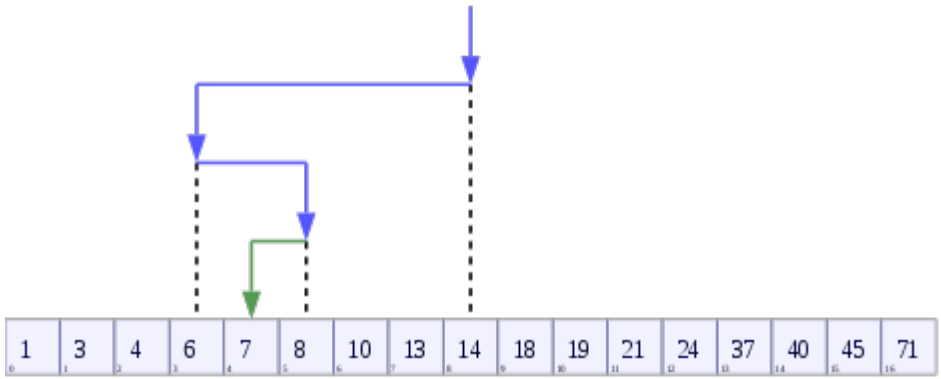
Algorithmus	**Worst-Case**	**Average-Case**
Heap-Sort	$O(n \log n)$	$O(n \log n)$
Merge-Sort	$O(n \log n)$	$O(n \log n)$
Quick-Sort	$O(n^2)$	$O(n \log n)$

Quick-Sort:

- Worst-Case bei Quicksort wird nur selten erreicht
- In der Praxis schneller als Merge-Sort und Heap-Sort

Suche in sortierten Listen

- Aufgabenstellung: Suchen eines Elementes in einer sortierten Liste (Array)
- Naiver Zugang: $O(n)$ Schritte (Erwartungswert $n/2$)
- Mittels binärer Suche: nur $O(\log n)$ Schritte notwendig
- Vorgehensweise: analog zu Suche in Telefonbuch
  - Beliebige Seite aufschlagen
  - Steht Name davor oder danach?
  - Weitere Suche erfolgt nur mehr im verbleibenden Teil



Speicherverbrauch, Konklusion

- Neben Laufzeitperformance ist auch der *Speicherverbrauch* relevant!
- Viele Algorithmen benötigen nur konstanten Speicherverbrauch $O(1)$
- Linearer Speicherverbrauch: zusätzlich $O(n)$ Speicher notwendig
- $O(n^2)$ oder $O(n^3)$ zusätzlicher Speicherverbrauch: funktioniert nur noch für kleine n
- Oft Tradeoff zwischen Laufzeiteffizienz und Speicherverbrauch

Konklusion:

- Bessere Hardware kann Performanceprobleme nur bei $O(n)$, oder evtl. $O(n \cdot \log n)$-Algorithmen beheben
- Bei "schwierigen" Problemen ist durch geschickte Programmierung (=Algorithmik) weitaus mehr zu erreichen

Euler-Zyklen

- Der Hierholzer-Algorithmus hat eine Laufzeit von $O(|E|)$.
- Die Laufzeit ist linear in der Anzahl der Kanten.
- Da $|E| \in O(|V|^2)$ kann die Laufzeit auch als quadratisch in der Anzahl der Knoten angesehen werden.
- Die Aussage $O(|E|)$ ist jedoch stärker, da viele Graphen nicht dicht besetzt sind.

Breitensuche und Tiefensuche

- Es werden alle Pfade zu allen Knoten betrachtet.
- Die Laufzeit ist somit $O(|V| + |E|)$.
- Die Laufzeit ist linear in der Anzahl der Kanten.
- Bei dicht besetzten Graphen führt dies jedoch zu $O(|V|^2)$.

Algorithmus von Dijkstra

Mit $n = |V|$ und $m = |E|$ können wir die Laufzeiteigenschaften angeben. Diese hängen von der konkreten Umsetzung der Prioritätswarteschlange Q ab.

Operation		Queue Implementierung		
Name	Anzahl	Liste	Heap	Fibonacci Heap[1]
decreaseKey	m	$O(1)$	$O(\log n)$	$O(1)$
getMin	n	$O(n)$	$O(\log n)$	$O(\log n)$
create	1	$O(n)$	$O(n)$	$O(n)$
Gesamt		$O(n^2 + m)$ $= O(n^2)$	$O((n+m)\log n)$	$O(n \log n + m)$

- Die Prioritätswarteschlange kann am einfachsten durch ein Array oder eine Liste umgesetzt werden. Bei jedem Aufruf von `getMin` muss die gesamte Liste durchlaufen werden um den kleinsten Eintrag zu ermitteln.
- Heap, bzw. Fibonacci-Heap sind baumbasierte Datenstrukturen, die wir jedoch nicht genauer besprechen.
- Diese führen jedoch zu besseren Laufzeitschranken (und wesentlich schnelleren Implementierungen)

[1] amortisierte Laufzeit

Algorithmus für die Starke Zusammenhangskomponente

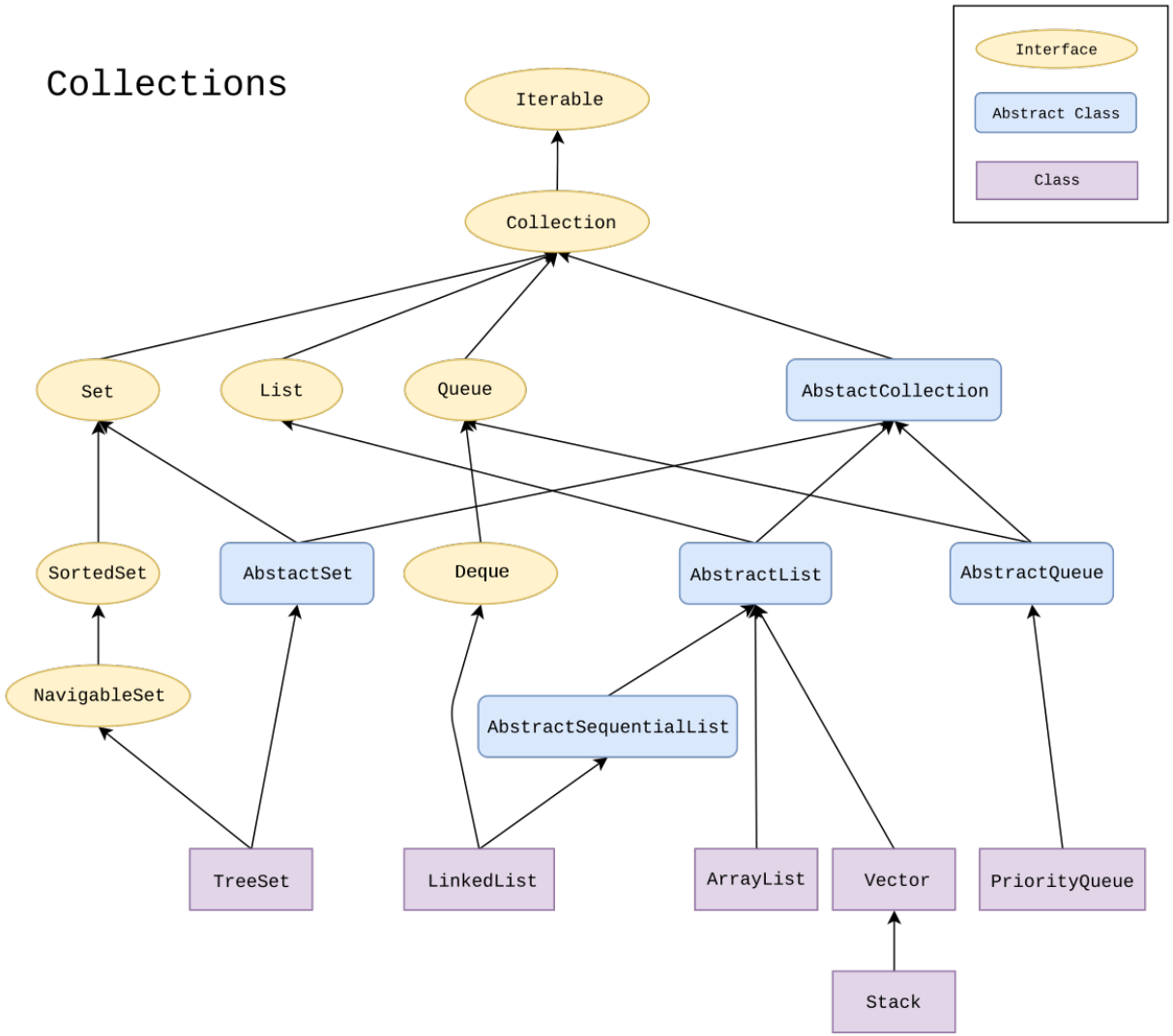
- Die Tiefensuche benötigt $O(|V| + |E|)$ Schritte.
- Die fertiggestellten Knoten können im Zuge der ersten Tiefensuche geordnet gespeichert werden, wodurch kein weiterer Aufwand entsteht.
- Somit ergibt sich der Gesamtaufwand durch zwei Tiefensuchen, und liegt somit insgesamt in $O(|V| + |E|)$.

Java Collections Framework

- Häufige Aufgabe in Programmen: Daten strukturiert abzuspeichern
- Das Java Collections Framework (JCF) enthält *wiederverwendbare* (generische) Klasse und Interfaces für diesen Zweck
- Die wesentliche Elemente der JCF sind: [2]
  - Geordnete Listen
  - Sets (Mengen)
  - Maps ("Dictionaries")
- Weitere Komponenten:
  - Stack
  - Queue

[2] In dieser Form in nahezu allen Programmiersprachen zu finden!

Java Collections Framework

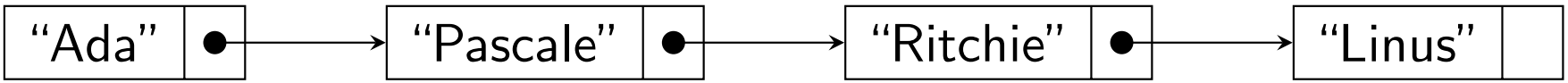


Arrays

- In Java bietet die Collection `ArrayList` einen komfortablen Umgang mit Arrays.
- Auf die Plätze des Arrays kann direkt mittels Index zugegriffen werden.
- Beim Einfügen werden dahinter liegende Elemtente verschoben.
- Beim Löschen werden dahinter liegende Elemtente nach vorne verschoben.

Verkettete Listen

- Verkettete Listen sind eine weitere Möglichkeit zur Umsetzung (Implementierung) des List-Interfaces (in Java).
- Die Collection in Java heißt `LinkedList` und bietet die gleichen Methoden wie `ArrayList`.
- Grundidee: jedes Element erhält einen Verweis ("Zeiger", "Link") auf den unmittelbaren Nachfolger.
- Wesentlicher Unterschied zu ArrayList: es gibt keine vorab definierten Plätze
- Um zum k-ten Element zu gelangen, müssen alle Elemente davor durchlaufen wernden!



Java: Set

- Das `Set`-Interface spezifiziert die Schnittstelle eines Sets
- Ein *Set* entspricht einer mathematischen Menge.
- Gleiche Elemente können in einem Set nur einmal enthalten sein.
- In einem Set kann man (im Gegensatz zur `ArrayList`) nicht mittels Index auf bestimmte Elemente zugreifen.
- Das `Set`-Interface wird konkret implementiert in der Klasse `TreeSet` und `HashSet`.

Java: HashSet

Wichtige Operationen:

- `HashSet()`: Erzeugt leeres `HashSet` mit Anfangskapazität 16
- `HashSet(int initialCapacity)`: Erzeugt Hashmap mit Anfangskapazität `initialCapacity`
- `boolean add(E e)`: Fügt Objektreferenz `e` hinzu
- `void clear()`: Löscht alle Elemente
- `boolean contains(Object o)`: Gibt an, ob Element `o` enthalten ist
- `boolean isEmpty()`: Gibt an, ob `HashSet` leer ist
- `boolean remove(Object o)`: Entfernt Objektreferenz `o`
- `int size()`: Gibt aktuelle Größe an

`HashSet`

- In einem `HashSet` werden die Elemente intern in einer *Hash-Table* gespeichert (umgesetzt mittels `HashMap`).
- Wir betrachten also zunächst das Konzept der **Hash-Tabelle** genauer.
- **Ziel beim Hashing ist es, auf bestimmte Objekte schnell zugreifen zu können.**
- Konkret: Zugriffszeit von $O(1)$ (also konstante Zeit), statt $O(n)$ bei Listen (wo bei der Suche alle Elemente durchlaufen werden müssen).

Hash-Tabelle

- Zu jeder abzuspeichernden Objektreferenz emittelt man zunächst ein *"Fach"*, in das es eingeschlichtet werden soll.

Beispiel: Wir definieren ein Ablagesystem für Objekte `Student`, so dass für jeden Anfangsbuchstaben des Nachnamens ein Fach angelegt wird. Bemerkung: In einem Fach können sich mehrere Studenten befinden, aber jeweils mit gleichem Anfangsbuchstaben des Nachnamens!

Hash-Tabelle

Warum benötigt man diese "Fächer" für das `Set`?

- Beim Hinzufügen eines Objektes muss geprüft werden, ob dieses nicht schon vorhanden ist.
- Durch das Fach-Konzept muss nur in diesem Fach gesucht werden, ob das Objekt schon vorhanden ist.
- **Deutlicher Verbesserung der Ausführungsgeschwindigkeit (Performance)!**

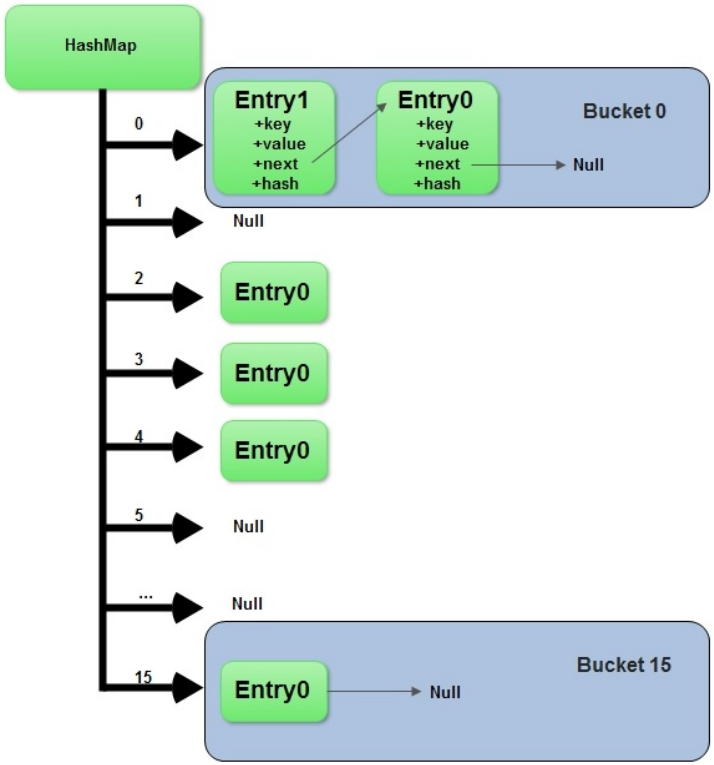
Hashfunktionen

- Hashfunktion bildet *(Schlüssel)*-Werte auf die Zielwerte (Hash-Werte) ab.
- Verschiedene Schlüssel können grundsätzlich den gleichen Hashwert haben $\Rightarrow$ **Kollision**, auch wenn dies nicht wünschenswert ist.
- Kollisionsbehandlung bei Hashtabellen:
  - Mehrere Objekte mit selbem Hashwert in Listen abspeichern
  - Sondieren: Berechnen alternativer Hashwerte
- Eine gute Hashfunktion liefert möglichst wenige Kollisionen!
- Kritischer Tradeoff bei Verwendung von Hash-basierten Datenstrukturen
  - Zu groß ("zuviele Fächer"): hoher Speicherverbrauch
  - Zu klein: viele Kollisionen $\Rightarrow$ Zu viele Elemente landen im selben "Fach" $\Rightarrow$ dadurch schlechtere Performance

- **Anmerkung:** Hashfunktion in der Kryptologie: zusätzliche Anforderung, dass Kollisionen gezielt gefunden werden können.

Verwendung von Hash-Tabellen

- Das Konzept der "Fächer" wird (intern) mit einer Hash-Tabelle (konkret: `HashMap`) umgesetzt.
- Die Fächer werden **Buckets** genannt.
- Für jedes Objekt muss nun ein Bucket berechnet werden!
- Diese Berechnung basiert auf dem Hash-Wert (Java: Methode `hashCode`).
- **Der Hash-Wert modulo der Anzahl der Buckets wird als Bucket-Index verwendet.**



- Grundsätzlich können verschiedene Objekte den selben Hash-Wert erhalten (und dadurch in das selbe Bucket eingeordnet werden).
- **Wichtig: Gleiche** Objekte **müssen** jedoch den gleichen Hash-Wert haben!
- Enthält ein Bucket mehr als eine Objektreferenz, so werden diese intern in einer `Liste` abgespeichert.

Java: `HashSet`

- Zur (korrekten!) Verwendung eines `HashSets` müssen wir also
  1. Hash-Codes zu Objekten berechnen, und
  2. die Gleichheit von Objekten ermitteln können
- Überschreiben der Methoden (aus `Object`):
  1. `int hashCode()`
  2. `boolean equals(Object o)`
- **Wichtig:** *Gleiche* Objekt **müssen** den gleichen Hash-Code haben!
  - Im `HashSet` wird zunächst nur der Hash-Code verglichen.
  - Nur wenn der Hash-Code gleich ist, wird die `equals`-Methode herangezogen

Beispiel (Java): `HashSet`

Beispiel:

- Klasse `Student` mit Attributen `name: String`, `alter: int`
- Verwendung in

```
1        HashSet<Student> students = new HashSet<>();
```

- Methoden `equals` und `hashCode` müssen in `Student` überschrieben werden!

Beispiel: HashSet

```
1  public class Student {
2    ...
3
4    @Override
5    public int hashCode() {
6      return name.charAt(0); // funktioniert, aber ist sehr unvorteilhaft (*)
7    }
8
9    @Override
10   public boolean equals(Object other) {
11     if (other == null) return false;
12     if (other == this) return true;
13     if (this.getClass() != other.getClass()) return false;
14
15     Student std = (Student)other;
16     if (this.name.equals(std.getName())
17         && this.alter == std.getAlter()))
18     {
19       return true;
20     }
21     return false;
22   }
23 }
```

Warum ist die Hashcode-Berechnung mittels erstem Buchstaben des Namens nicht ideal?

Beispiel: HashSet

```
1  public class Student {
2    ...
3
4    @Override
5    public int hashCode() {
6      return name.charAt(0); // funktioniert, aber ist sehr unvorteilhaft (*)
7    }
8
9    @Override
10   public boolean equals(Object other) {
11     if (other == null) return false;
12     if (other == this) return true;
13     if (this.getClass() != other.getClass()) return false;
14
15     Student std = (Student)other;
16     if (this.name.equals(std.getName())
17         && this.alter == std.getAlter()))
18     {
19       return true;
20     }
21     return false;
22   }
23 }
```

Warum ist die Hashcode-Berechnung mittels erstem Buchstaben des Namens nicht ideal? Es werden nur 26 verschieden Hashcodes berechnet. Bei einer höheren Bucketanzahl in Hashtabellen werden dadurch die meisten nicht benutzt. Eine gleichmäßige Verteilung der Objekte in die Buckets ist somit nicht gewährleistet.

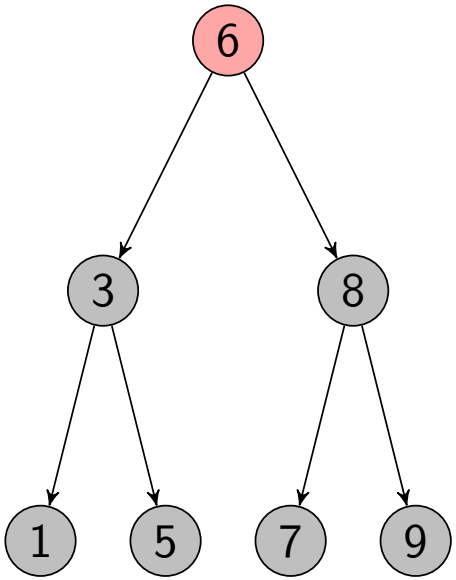
Set mittels Bäumen (Java: `TreeSet`)

- Umsetzung des *Set*-Konzeptes mittels *Bäumen*.
- Konkret: **binäre Suchbäume**.
- Man nennt einen (Wurzel-)Baum *Binärbaum* wenn jeder Knoten höchstens zwei Nachfolger (=“Kinder”, “Kind-Knoten”) hat.
- In einem Suchbaum gilt für jeden Knoten: alle Elemente[3] mit kleineren Werten sind im linken Teilbaum zu finden, jene mit größeren im rechten.
- Ein Binärbaum heißt *vollständig*, wenn alle “Ebenen” bis auf die letzte Ebene vollständig gefüllt sind.
- Ein Binärbaum heißt *voll*, wenn jeder Knoten entweder ein Blatt ist, oder genau zwei Kinder besitzt.
- Binäre Suchbäume weisen günstige Eigenschaften zum Speichern von Elementen auf, wenn Elemente schnell gefunden werden sollen.
- Der wesentliche Grund dafür ist, dass annähernd volle und vollständige Bäume mit n Elementen nur $O(\mathrm{ld}\, n)$ viele Ebenen besitzen.
- Man nennt die Anzahl der Ebenen auch die **Höhe** des Baumes.

[3] in diesem Teilbaum

(Binärer) Suchbaum

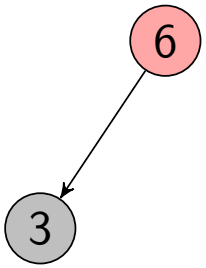
- Achtung: in der Darstellung ist in den Knoten der *Inhalt* (also ein `int`-Wert) dargestellt (und nicht der Knotenname, bzw. Index)



- Binäre Suchbäume:
  - Ein vollständiger voller Binärbaum mit Höhe d hat $2^d - 1$ Knoten
  - Beispiel: Höhe $3 \Rightarrow 2^3 - 1 = 7$ Knoten
  - Umgekehrt hat ein vollständiger und voller Binärbaum mit n Knoten eine Höhe von $O(\mathrm{ld}\ n)$

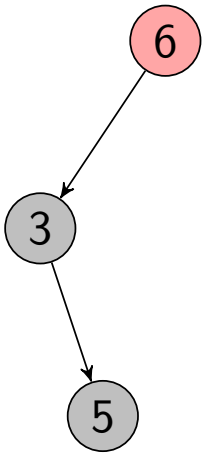
- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:
  - Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



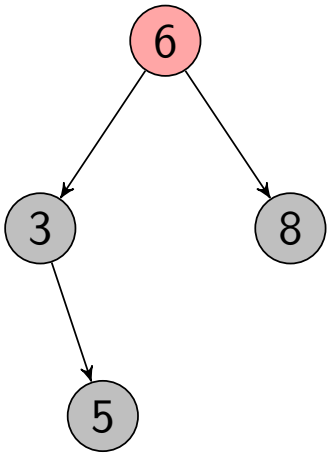
- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



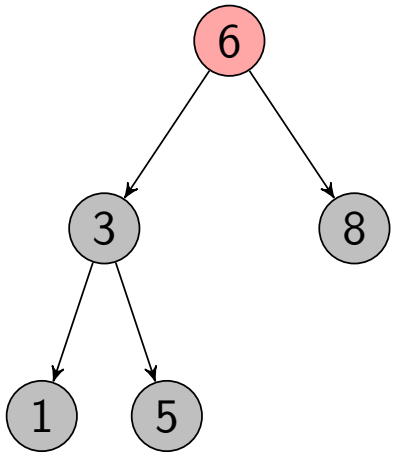
- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



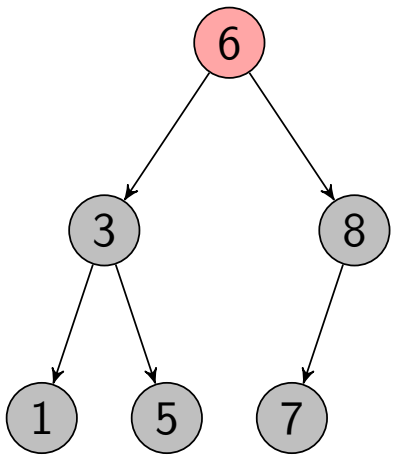
- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



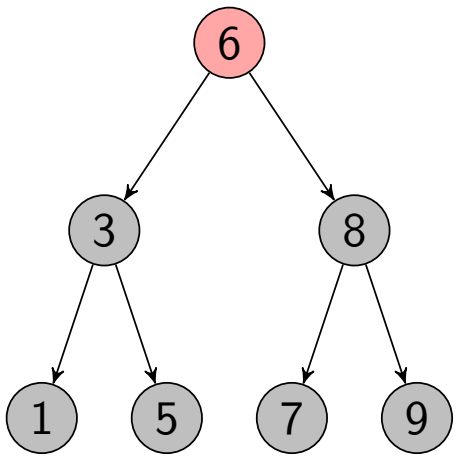
- Das erste Element "6" wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger ("Kind") der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!
- 1 wird im linken Teilbaum am ersten freien Platz eingefügt, also als linkes Kind von 3.

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



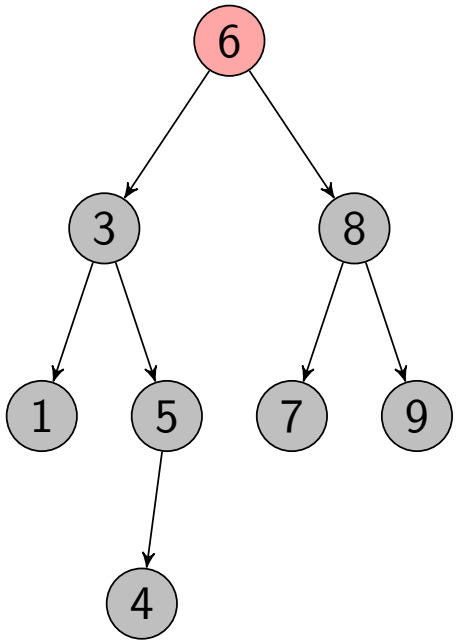
- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!
- 1 wird im linken Teilbaum am ersten freien Platz eingefügt, also als linkes Kind von 3.
- 7 kommt in den rechten Teilbaum, als linkes Kind von 8

- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!
- 1 wird im linken Teilbaum am ersten freien Platz eingefügt, also als linkes Kind von 3.
- 7 kommt in den rechten Teilbaum, als linkes Kind von 8
- 9 wird als rechtes Kind von 8 eingefügt.

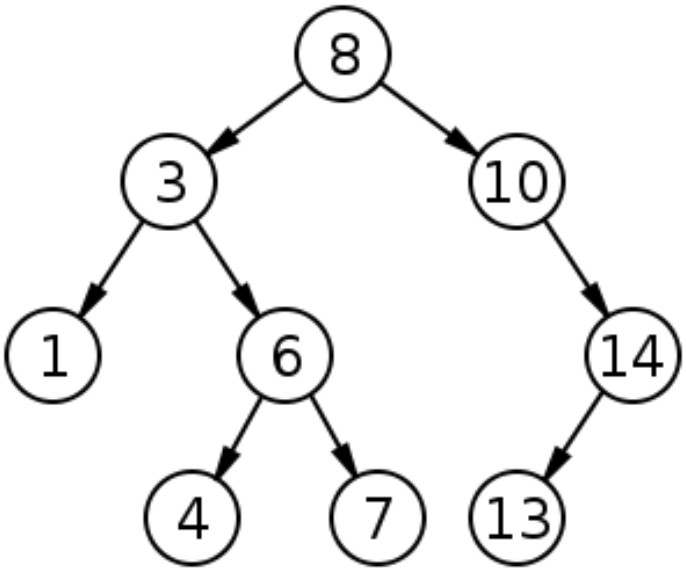
- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



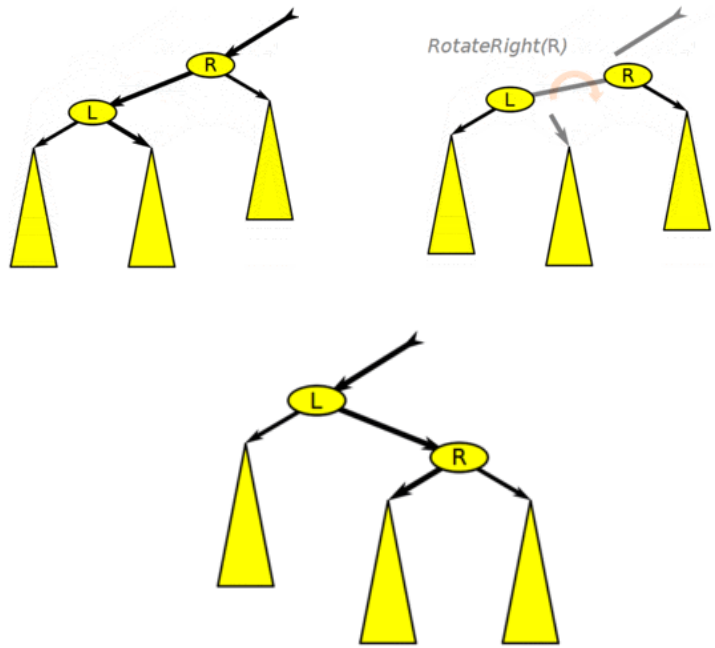
- Das erste Element “6” wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger (“Kind”) der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!
- 1 wird im linken Teilbaum am ersten freien Platz eingefügt, also als linkes Kind von 3.
- 7 kommt in den rechten Teilbaum, als linkes Kind von 8
- 9 wird als rechtes Kind von 8 eingefügt.
- 4 wird in einer neuen Ebene als linkes Kind von 5 eingefügt.

Balancierte (binäre) Suchbäume

Binärer Suchbaum:



Balancierte Suchbäume:



Bildquelle: Wikipedia

- In vollständigen binären Suchbäumen können Elemente in maximal $O(\mathrm{ld}n)$ Schritten gefunden werden!
- Balancierung muss gewährleistet werden indem der Baum regelmäßig ausbalanciert wird.
- Konkrete Implementierungen sind z.B. Red-Black-Trees oder AVL-Trees.

TreeSet

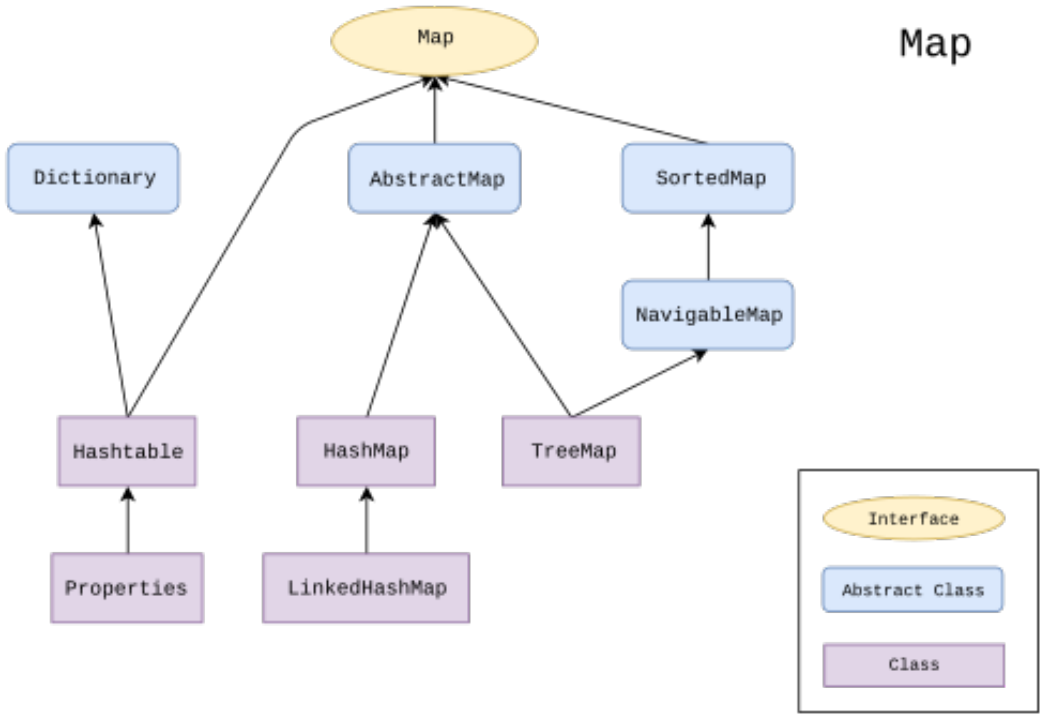
- `TreeSet` bietet langsamere elementare Operationen: `add`, `remove`, `contains`
- Dafür ist schnelleres sortiertes Durchlaufen der Elemente möglich
- **Bei der Verwendung von `TreeSet<E>` muss `E` das Interface `Comparable<E>` implementieren!** [4]

Wichtige Operationen (zusätzlich zu `HashSet`):

- `E first()`: liefert kleinstes Element
- `E last()`: liefert größtes Element
- `E higher(E e)`: liefert nächst größeres Element
- `E lower(E e)`: liefert nächst kleineres Element

[4] Alternativ: `Comparator`

Java: `Map`-Klassenhierarchie



Bildquelle: Wikipedia

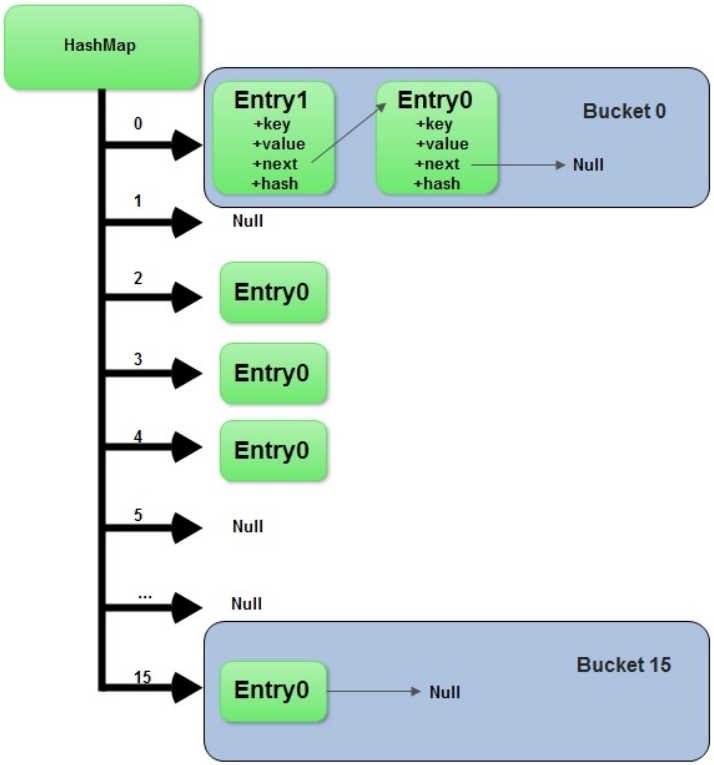
Map

- Maps (oft auch "dictionaries") speichern **Schlüssel – Wert**-Paare.
- Oft möchte man gespeicherte Objekte *schnell* aufgrund eines *Schlüssels* (id, name, ...) finden, bzw. darauf zugreifen.
- Beispiel:
  - Finde Mitarbeiter mit Personalnummer
  - Finde Immobilie mittels Anzeigennummer
- Das Suchen in Arrays, ArrayLists etc. ist aufwändig, da die Datenstruktur zunächst durchsucht werden muss.
- **Ziel:** Auffinden eines Elementes aufgrund des **Schlüssels (Key)** in (quasi) konstanter Zeit $O(1)$, d.h. ohne viele Elemente in der Datenstruktur betrachten zu müssen.
- Die `HashMap<K, V>` bietet diese Funktionalität an
  - K ... Key
  - V ... Value

HashMap

Funktionsweise:

- Berechnung eines *Hash-Wertes* aus dem Schlüssel (Key)
  - Diese Berechnung erfolgt im Allgemeinen sehr schnell (z.B. im Vergleich zum Durchsuchen eines Arrays)
- Der Hash-Wert wird als Basis für die Index-Berechnung (Buckets) verwendet.
- Zu jedem Schlüssel (Key) wird *genau ein* Wert abgespeichert.



HashMap

Wichtige Operationen:

- `HashMap()`: Erzeugt Hashmap mit 16 Buckets
- `HashMap(int initialCapacity)`: Erzeugt Hashmap mit `initialCapacity` Buckets
- `V put(K key, V value)`: Fügt ein **Schlüssel-Wert-Paar** hinzu. Ein gegebenenfalls zuvor vorhandener Wert mit diesem Schlüssel wird zurückgegeben.
- `V get(Object key)`: Liefert Wert mit entsprechendem Schlüssel, sofern vorhanden.
- `isEmpty()`: Gibt an ob die Hashmap leer ist.
- `Set<K> keySet()`: Liefert alle verwendeten Schlüssel

Beispiel: HashMap

Einfügen und Auslesen von Objektreferenzen:

```
1  HashMap<Integer, Student> studenten = new HashMap<>();
2  Student s1 = new Student("Peter", 20);
3  Integer id = s1.getId();
4  studenten.put(id, s1);
5
6  ...
7  Student s2 = studenten.get(id);
```

Durch die Werte einer HashMap iterieren:

```
1  Iterator<Student> it = studenten.entrySet().iterator();
2  while (it.hasNext()) {
3    Map.Entry<Integer, Student> pair = (Map.Entry)it.next();
4    System.out.println(pair.getKey() + ":␣" + pair.getValue());
5  }
```

Verwendung Hash- und Tree-Collections

Bei der Verwendung dieser Collections ist zu beachten:

- `HashSet<T>`, `HashMap<K, V>`:
  - Implementierung von `boolean equals(Object o)` in Klassen `T` bzw. `K`
  - Implementierung von `int hashCode()` in Klassen `T` bzw. `K`
- `TreeSet<T>`:
  - Klassen `T` muss interface `Comparable<T>` implementieren

Laufzeitananlyse

Laufzeitanalyse verschiedener Operationen bei Größe n der Collection:

Operation	`ArrayList`	`LinkedList`	`HashSet`	`TreeSet`
add (end)	$O(1)$	$O(1)$	$O(1)$	$O(\mathrm{ld}\ n)$
remove	$O(n)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
remove (iterator)	$O(n)$	$O(1)$	$O(1)$	$O(\mathrm{ld}\ n)$
get (index)	$O(1)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
access (position)	$O(1)$	$O(n)$		
find (element)	$O(n)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
insert (position)	$O(n)$	$O(n)$		
insert (iterator)	$O(n)$	$O(1)$		

Anmerkung: In der Tabelle scheint das `HashSet` dem `TreeSet` eindeutig überlegen zu sein. Das `TreeSet` bietet jedoch den Vorteil, dass die enthaltenen Elemente effizient sortiert ausgegeben werden können, und es benötigt wesentlich weniger zusätzlichen Speicher als das `HashSet`.