Contents

1 Datenstrukturen

1.1 ArrayList

Die ArrayList in Java ist eine dynamisch wachsende Liste, die auf dem zugrunde liegenden Array basiert. Sie erweitert die Funktionalität von Arrays durch automatische Größenanpassung und bietet eine flexible Möglichkeit, Elemente hinzuzufügen, zu löschen und darauf zuzugreifen.

Anwendungsfälle:
- Speicherung von Elementen einer Liste
- Zugriff auf Elemente durch Index
- Wenn die Größe der Liste nicht im Voraus bekannt ist
- Wenn häufiger Einfüge- und Löschoperationen erforderlich sind

Syntax:

```java
// Import der ArrayList-Klasse
import java.util.ArrayList;

// Deklaration und Initialisierung
ArrayList<Integer> myArrayList = new ArrayList<>();

// Elemente hinzufuegen
myArrayList.add(1);

// Zugriff auf ein Element
int element = myArrayList.get(0);

// Iteration durch die ArrayList
for (int i = 0; i < myArrayList.size(); i++) {
    System.out.println(myArrayList.get(i));
}
```

Funktionalität unter der Haube:
- Intern durch Array verwaltet, welches ggf. neu dimensioniert werden muss
- Neudimensionierung führt zu O(n), da neues Array und alle Elemente kopiert werden. Wird allerdings selten gemacht und dann i.d.R. direkt verdoppelt, somit amortisiert O(1)

Laufzeitkomplexität

Operation	Add	Remove	Get	Contains	Next	Size
Time Complexity	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$

1.2 LinkedList

Dynamische Datenstruktur bestehend aus Knoten, die jeweils auf den nächsten Knoten in der Sequenz zeigen Zwei Haupttypen: Singly Linked (Zeigt nur auf Nachfolger), Doubly Linked (Zeigt auf Vor- und Nachfolger).

Anwendungsfälle:
- Bei häufigen Einfüge- und Löschoperationen an verschiedenen Positionen
- Wenn die Größe der Liste variabel ist

Syntax:

```java
// Import der ArrayList-Klasse
import java.util.ArrayList;

// Deklaration und Initialisierung
ArrayList<Integer> myArrayList = new ArrayList<>();

// Elemente hinzufuegen
myArrayList.add(1);

// Zugriff auf ein Element
int element = myArrayList.get(0);

// Iteration durch die ArrayList
for (int i = 0; i < myArrayList.size(); i++) {
    System.out.println(myArrayList.get(i));
}
```

Funktionalität unter der Haube:
- Jeder Knoten erhält Verweise auf Nach- (ggf. Vorgänger)
- Einfügen und Löschen an am Ende/Anfang konstant, ansonsten linear, da update der Verweise

Laufzeitkomplexität

Operation	Add	Remove	Get	Contains	Next	Size
Time Complexity	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

1.3 Stack

Agiert nach **L**ast-**I**n-**F**irst-**O**ut (**LIFO**) Prinzip: Das letzte hinzugefügte Element wird als Erstes entfernt. Unterstützt lediglich Hinzufügen (Push) und Entfernen (Pop).

Anwendungsfälle:
- Rückverfolgung von Funktionsaufrufen (Call Stack Aufbau bspw.)
- Umkehrung von Zeichenketten/Ausdrücken
- Undo-Mechanismen

Syntax:

```java
// Verwendung des Stack-Datentyps
import java.util.Stack;

// Deklaration und Initialisierung
Stack<Integer> myStack = new Stack<>();

// Elemente hinzufuegen
myStack.push(1);

// Element entfernen
int poppedElement = myStack.pop();
```

Funktionalität unter der Haube:
- Ein Stack kann durch ein Array oder eine verkettete Liste implementiert werden
- Die Elemente werden auf dem Stack oben hinzugefügt und entfernt
- Schnelle Hinzufüge- und Entfernungsoperationen (konstante Laufzeit O(1))
- Keine Suchmöglichkeit, ggf. aber Möglichkeit oberstes Element anzuschauen (peek)
- Custom Suche implementierbar, dann O(n) wahrscheinlich

Laufzeitkomplexität

Operation	Push	Pop
Time Complexity	$O(1)$	$O(1)$

1.4 Queue

Agiert nach **F**irst-**I**n-**F**irst-**O**ut (**FIFO**) Prinzip: Das letzte hinzugefügte Element wird als Erstes entfernt. Unterstützt lediglich Hinzufügen (Enqueue) und Entfernen (Dequeue)

Anwendungsfälle:
- Warteschlangen in Betriebssystemen für Prozessplanung
- Puffer (Bspw. Datenverarbeitung)
- Aufgabenwarteschlangen (Bspw. in Multi-Thread-Anwendungen)

Syntax:

```java
// Verwendung des Queue-Datentyps
import java.util.LinkedList;
import java.util.Queue;

// Deklaration und Initialisierung
Queue<Integer> myQueue = new LinkedList<>();

// Elemente hinzufuegen
myQueue.offer(1);

// Element entfernen
int dequeuedElement = myQueue.poll();
```

Funktionalität unter der Haube:
- Eine Queue kann bspw. durch ein Array oder eine verkettete Liste implementiert werden
- Die Elemente werden auf der einen Seite (Rückseite) hinzugefügt und auf der anderen Seite (Vorderseite) entfernt
- Schnelle Hinzufüge- und Entfernungsoperationen (konstante Laufzeit O(1))
- **Spezialfall**: Double Ended Queue ermöglicht Operationen an beiden Seiten
- **Spezialfall**: Priority Queue, wo Elemente nach Priorität in die Queue eingefügt werden (Über Comparator Interface umzusetzen)

Laufzeitkomplexität (Anhand LinkedList-basierter Queue)

Operation	Offer	Peak	Poll	Remove	Size
Time Complexity	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

1.5 HashMap

Assoziative Datenstruktur, speichert Key-Value Paare Erlaubt extrem schnelles Suchen, Einfügen und Löschen durch Hash-Verfahren

Anwendungsfälle:
- Effiziente Suche nach Werten in großen Datenmengen
- Implementierung von Caches
- Verfolgung von Zählungen oder Häufigkeiten

Syntax:

```java
// Verwendung des HashMap-Datentyps
import java.util.HashMap;

// Deklaration und Initialisierung
HashMap<String, Integer> myHashMap = new HashMap<>();

// Werte hinzufuegen
myHashMap.put("One", 1);

// Wert abrufen
int value = myHashMap.get("Two");
```

Funktionalität unter der Haube:
- Implementiert häufig eine Hashtabelle, die Schlüssel in Indizes umwandelt, um Werte zu speichern und abzurufen
- Verwendet Hashfunktionen, um Kollisionen zu minimieren und den Zugriff auf die Werte zu beschleunigen
- Schnelle Einfüge-, Lösch- und Suchoperationen (konstante Laufzeit O(1)), aber bei Kollisionen (Zwei versch. Elemente, gleicher Hash) kann die Leistung abnehmen

Laufzeitkomplexität

Operation	Get	ContainsKey	Next
Time Complexity	$O(1)$	$O(1)$	$O(h/n)$

1.6 Set

Ungeordnete Sammlung eindeutiger Elemente, somit keine Duplikate erlaubt

Anwendungsfälle:
- Überprüfen auf Eindeutigkeit von Elementen
- Durchlaufen und Verarbeiten von eindeutigen Elementen
- Mengenoperationen wie Vereinigung, Schnittmenge und Differenz

Syntax:

```java
// Verwendung des HashMap-Datentyps
import java.util.HashMap;

// Deklaration und Initialisierung
HashMap<String, Integer> myHashMap = new HashMap<>();

// Werte hinzufuegen
myHashMap.put("One", 1);

// Wert abrufen
int value = myHashMap.get("Two");
```

Funktionalität unter der Haube:
- Implementiert häufig eine Hashtabelle oder andere effiziente Mechanismen zur Überprüfung auf Eindeutigkeit
- Verwendet Hashfunktionen, um schnelles Hinzufügen, Löschen und Überprüfen auf Eindeutigkeit zu ermöglichen
- Schnelle Operationen für Mengenoperationen

Laufzeitkomplexität (Hash Table based)

Operation	Add	Remove	Contains	Next	Size
Time Complexity	$O(1)$	$O(1)$	$O(1)$	$O(h/n)$	$O(1)$

1.7 Tree

Hierarchische Datenstruktur bestehend aus Knoten, die durch Kanten verbunden sind Verschiedene Arten von Bäumen: Suchbäume, AVL-Bäum, Heaps,..

Anwendungsfälle:
- Organisation von Daten in hierarchischen Strukturen
- Schnelle Suche, Einfügung und Löschung von Elemente

Syntax:

```
// Beispiel f r einen einfachen bin ren Baum
// Muss man selbst bauen bzw Libraries nutzen
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    public TreeNode(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}
```

Funktionalität unter der Haube:
- **Einfügen (Hinzufügen):**
- Abhängig von der Art des Baumes, erfolgt das Hinzufügen von Elementen durch Traversieren und Einfügen an der richtigen Stelle.
- Bei binären Suchbäumen wird das Element entsprechend der Ordnungsregel eingefügt.
- Bei AVL-Bäumen erfolgt zusätzlich eine Balance-Anpassung, um die Höhe des Baumes zu minimieren und somit die Suche zu optimieren.

- **Löschen (Entfernen):**
- Entfernen eines Elements erfordert ebenfalls eine Traversierung, um das Element zu finden.
- Je nach Art des Baumes müssen nach dem Entfernen gegebenenfalls Anpassungen vorgenommen werden, um die Baumstruktur beizubehalten.
- AVL-Bäume werden nach dem Löschen ebenfalls balanciert.

- **Suchen (nach einem Element):**
- Suche erfolgt durch Traversieren des Baumes entsprechend der Ordnungsregel.
- Bei einem binären Suchbaum wird die Suche abhängig von der Vergleichsregel im Baum durchgeführt

- **Durchlaufen aller Elemente:**

- Die Traversierung kann in-order, pre-order oder post-order erfolgen, je nach Anforderungen.
- *In-order*: Linker Teilbaum, aktueller Knoten, rechter Teilbaum.
- *Pre-order*: Aktueller Knoten, linker Teilbaum, rechter Teilbaum.
- *Post-order*: Linker Teilbaum, rechter Teilbaum, aktueller Knoten.

- **Spezialfall Min-Heap:**
- Das kleinste Element befindet sich an der Wurzel
- Jeder Elternknoten ist kleiner als oder gleich seinen Kindern
- Verwendung in der Implementierung von Prioritätswarteschlangen, wo das Element mit der höchsten Priorität (niedrigster Wert) immer an der Spitze steht

- **Spezialfall Max-Heap:**
- Das größte Element befindet sich an der Wurzel
- Jeder Elternknoten ist größer als oder gleich seinen Kindern
- Auch in der Implementierung von Prioritätswarteschlangen verwendet, jedoch mit der höchsten Priorität an der Spitze

- **Spezialfall AVL-Baum:**
- Ein ausbalancierter binärer Suchbaum, der sicherstellt, dass die Höhen der beiden Kindunterbäume jedes Knotens höchstens um eins voneinander abweichen
- Die Balance wird durch Rotationen (links oder rechts) bei Einfüge- oder Löschoperationen aufrechterhalten
- Die Balancierung sorgt für effiziente Suchoperationen, da die Höhe des Baumes minimiert wird

Laufzeitkomplexität

Operation	TODO	TODO	TODO
Time Complexity	*TODO*	*TODO*	*TODO*

1.8 Graph

Sammlung von Knoten (Vertices) und Kanten (Edges), welche Knoten verbinden. Kann gerichtet (Directed) oder ungerichtet (undirected sein)

Anwendungsfälle:
- Modellierung von Beziehungen in sozialen Netzwerken
- Routing-Algorithmen in Computernetzwerken
- Suche in Graphen, wie in der Tiefen- oder Breitensuche

Syntax:

```
class Graph {
    // TODO
}
```

Funktionalität unter der Haube:
- Stark variierend je nach Umsetzung

Laufzeitkomplexität

Operation	TODO	TODO	TODO
Time Complexity	*TODO*	*TODO*	*TODO*