Konzepte

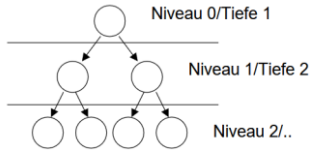
Queue
FIFO (first in first out)
enqueue, dequeue, isEmpty
Auch möglich mit priority

LinkedList
Schnelle Änderung, langsamer Zugriff. Am Anfang einfügen O(1), Schluss O(n)
Bei Doppelt verkettet prev und next.

Sortiere Listen
Implementieren Comparable

Rekursion
Basisfall: Der Fall, in dem die Rekursion stoppt
Rekursiver Fall: Der Fall, in dem der Algorithmus sich selbst aufruft, mit einem kleineren Problem.

Binärbäume
Knoten: 2^n
Max: 2^{k-1}
Vollständig: Alle Ebenen ausser letzten voll



Preorder Knoten zuerst: n, A, B
Inorder Knoten in der Mitte: A, n, B
Postorder Knoten am Schluss: A, B, n
Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ..

Beim Löschen von Fall 3:
vom linken Teilbaum wir das Element
ganz rechts als Ersatz genommen

Stacks
LIFO (last in first out)
push, pop, isEmpty

Arrays
Zugriff ist effizient, jedoch verändern nicht.

Liskovsches Substitutionsprinzip
Wenn T verwendet, funktioniert auch korrekt mit abgeleiteter Klasse von T

Type Erasure
Generische Typen werden zur Laufzeit entfernt (Kompatibilität mit älterem Code)

Sortiert Binärbäume
Für jeden Knoten im Baum gilt die **Invariante**
im linken Unterbaum sind alle kleineren Elemente K_L <=* k
im rechten Unterbaum sind alle grösseren Elemente: K_R >* k

Beim Einfügen muss links eingefügt werden, wenn das neue Element kleiner oder gleich ist, sonst rechts

Suche:
Wenn x == Wurzelelement gilt, haben wir x gefunden.
Wenn x > Wurzelelement gilt, wird die Suche im rechten Teilbaum von B fortgesetzt, sonst im linken Teilbaum.

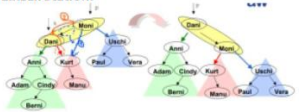
Gleich aufwändig wie Binarysearch, Log2 Schritte

Balanced Binärbäume
Voller Baum: Alles bis auf letzte Stufe gefüllt.
Schlimmster Fall: Liste
Vollständig ausgeglichen: Die Gewichte der beiden Teilbäume unterscheiden sich max um 1.
AVL-Ausgeglichenheits-Bedingung:
- Tiefen unterscheiden sich max. um 1
- Beim Einfügen und Löschen muss diese Bedingung eingehalten bleiben

⇨ Einfacher als Gewichtsbedingung
⇨ Suchoperationen: O(log(n))

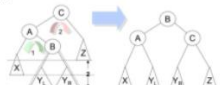
Wichtig: AVL-Bäume sind sortiert.
- Pro Knoten eine Zahl, die speichert, wie tief die nachfolgenden Teilbäume sind.

Einzelrotation:



⇨ Problem: Wenn ein Teilbaum Tiefe > 1 (=2) hat kann nicht mit Einzelrotation balanciert werden.

Doppelrotation:



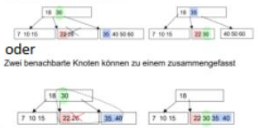
B-Bäume
Entwickelt, um mit Bäume effizient auf Festplatten oder anderen sekundären Speichermedien zu arbeiten (mittels «Schlüssel»/«Index»)
Idee: Zugriffe auf Blöcke minimieren.
- Baum immer ausgeglichen
- Möglichst viele Infos in einem Block
- Möglichst breiter Baum
- Alle Knoten gleich gross

Bedingungen:
- B-Baum mit Ordnung n enthält jeder Knoten ausser der Wurzel min. n/2max.n-1 Schlüssel
- B-Bäume werden automatisch balanciert
- Jeder Knoten ist entweden ein Blatt oder hat m+1 Nachfolger (m = Anzahl Schlüssel des Knotens)
- Alle Schlüssel sortiert (innerhalb Knoten)
- Alle Blätter auf selber Stufe
- Mind. n/2 Unterbäume Baum=weniger hoch

Einfügen:
⇨ Immer in den Blättern.
- Falls Platz: Einfügen
- Falls Überlauf:

⇨ Extrem Fall: Vaterknoten überlauft (geht bis zur Wurzel => Tiefe + 1)

Löschen:
- Falls Blatt: Element Löschen
- Falls innerer Knoten: Gleich wie Binärbaum (Ersatzwert suchen: Rechtestes Element im linken Teilbaum)
- Falls Blatt und Unterlauf:



Suchen:
⇨ Ähnlich wie Binärbaum:
1) den Wurzelblock lesen
2) gegebenen Schlüssel S auf dem gelesenen Block suchen
3) wenn gefunden, Datenblock lesen fertig
4) ansonsten i finden, sodass $S_i < S < S_{i+1}$
5) Block Nr i einlesen, Schritte 2 bis 5 wiederholen

Tiefe des Baumes $\lceil \log_{\text{Anzahl Verweise}} \text{Anzahl Elemente} \rceil$

Anzahl Zugriffe: proportional zu Tiefe des Baumes

2-3-4: Max 4 Nachfolger
Rot-Schwarz: Alle Pfade von der Wurzel zu einem Blatt enthalten gleich viele schwarze Knoten. Die Wurzel ist schwarz. Jedes Blatt ist schwarz. Kein roter Knoten hat ein rotes Kind

Graph
Komplett: Jeder Knoten ist mit jedem anderen verbunden.
Dicht: Fast alle möglichen Kanten existieren.
Dünne: Relativ wenige Kanten im Vergleich zur Anzahl der möglichen Kanten.
Datenstrukturen:
1. **Adjazenzliste**
   - Jeder Knoten führt eine Liste der Nachbarknoten.
   - Speicherplatz: $O(n+m)$ (n = Knotenanzahl, m = Kantenanzahl).
   - Gut für dünne Graphen.
2. **Adjazenzmatrix**
   - Boolean- oder Gewichts-Matrix mit Einträgen für Kantenverbindungen.
   - Speicherplatz: $O(n^2)$.
   - Effizient für dichte Graphen, jedoch speicherintensiv.

Traversierungen:
Tiefensuche (DFS):
- Erkundet die Knoten, indem es tiefer geht, bevor es Nachbarn besucht.
- Pseudocode verwendet einen Stack.
- Anwendung: Zykluserkennung, Topologische Sortierung.

Breitensuche (BFS):
- Besucht alle Knoten eines Levels, bevor es tiefer geht.
- Pseudocode verwendet eine Queue.
- Anwendung: Kürzester Pfad in ungewichteten Graphen.

Adjazenzmatrix: Platzbedarf $O(n^2)$.

Adjazenzliste: Platzbedarf $O(n+m)$.

TSP: Exponentielle Laufzeit $O(n!)$.

DFS: Preorder, BFS: Levelorder

Aufwände
Sortierter Array:
- Einfügen O(n/2)
- binäres Suchen O(log2 (n))

Lineare sortierte Liste:
- Einfügen O(n/2)
- Suchen O(n/2)

Sortierter Binärbaum:
- Einfügen O(log2 (n))
- Suchen O(log2 (n))

Raw Types
Abwärtskompabilität.
Problem: Begrenzte Typensicherheit zur Laufzeit.

Bounds

Bound	Verwendung
Upper Bound (`extends`)	Typ ist `X` oder eine Unterklasse davon
Lower Bound (`super`)	Typ ist `X` oder eine Oberklasse davon
Multiple Bounds (`&`)	Kombination von Klasse + Interfaces

Beispiel

```
T extends Number
? super Integer
T extends Number & Comparable<T>
```

Unterg: Oberklasse
Oberg: Unterklasse

Regex

Metasymbol	Beispiel	Bedeutung	Menge der gültigen Literale
*	ax*b	0 oder mehrere x	ab, axb, axxb, axxxb, ...
+	ax+b	1 oder mehrere x	axb, axxb, axxxb, ...
?	ax?b	x optional	ab, axb
|	a|b	a oder b	a, b
()	x(a|b)x	Gruppierung	xax, xbx
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
[]	[abc]x	1 Zeichen aus einer Menge	ax, bx, cx
[-]	[a-h]	Zeichenbereich	a,b,c, ..., h
\d	\d\d	Digit[0-9]	78, 10
\D	\D	kein Digit	a, b , c
^	^\d	Negation	a, b , c
\s	\s	Leerzeichen (Blank,etc)	blank, tab, cr,
\S	\S	kein Leerzeichen	

Algorithmen

Dijkstra

Berechnet den kürzesten Pfad in einem gewichteten Graphen mit positiven Kanten.

3 Gruppen:

- Besuchte Knoten
- Benachbarte Knoten
- Unbesehene Knoten (n.prev == null)

O(n^2) (Im schlimmsten Fall)

Setze Startknoten in Queue.

Solange Queue nicht leer ist:

- Nimm erstes Node aus Queue
- Markiere dieses
- Wenn = goal: Fertig
- Für alle Kanten aus Node:
  - Wenn nicht markiert &(kleinere Distanz als bereits besucht oder unbesehen):
    - Berechne Distanz
    - Setze Distanz und prev
    - Lege in Queue mit Distanz als Priority

Maximaler Fluss

Beschreibung:

- Berechnet den maximalen Fluss in einem Flussnetzwerk.
- Nutzt Restgraphen und augmentierende Pfade.
- Was in einen Knoten hineinfließt, muss auch hinausfließen.

Lösungsschritte:

1. Finde einen augmentierenden Pfad im Restgraphen.
2. Aktualisiere den Fluss entlang des Pfades.
3. Wiederhole, bis kein augmentierender Pfad mehr existiert.

Zielfunktion

Ziel ist es, f(v) zu maximieren (bei Gewinnstrategien) oder zu minimieren (bei Kostenoptimierung).

Beispiel Labyrinth:

$f(v)$ = bisherige Weglänge + geschätzte Entfernung zum Ziel (Luftlinie).

Minimaler Spannbaum

Findet den Spannbaum mit minimalen Gesamtkosten in einem gewichteten Graphen. Ähnlich wie der Dijkstra-Algorithmus:

1. Beginne mit einem Knoten.
2. Füge iterativ die Kante mit minimalen Kosten hinzu, die einen neuen Knoten verbindet.

Topologische Sortierung

Ordnet die Knoten eines azyklischen gerichteten Graphen in eine lineare Reihenfolge, sodass Abhängigkeiten berücksichtigt werden.

Algorithmus Idee:

1. Zähle die eingehenden Kanten für jeden Knoten.
2. Wähle einen Knoten ohne eingehende Kanten und füge ihn zur Sortierung hinzu.
3. Entferne die ausgehenden Kanten dieses Knotens.
4. Wiederhole, bis alle Knoten sortiert sind.

O(V+E) (mit V Knoten und E Kanten)

Springer etc

Springer(alle besuchen), 8 Damen(keine schlagen), Rucksack(K Gegenstände M grösse)

Quicksort

Partitionierung: Verschiebe alle Elemente auf die richtige Seite des Pivots

Pivot: Wichtig für die Effizienz das es in der Mitte liegt:

Methode	Vorteil	Nachteil
Erstes Element	Einfach zu implementieren	Schlecht für sortierte Listen
Letztes Element	Einfach zu implementieren	Schlecht für sortierte Listen
Mittleres Element	Reduziert schlechte Partitionierung	Nicht optimal
Zufälliges Element	Reduziert schlechtesten Fall	Nicht deterministisch
Median von Drei	Stabil und effizient	Etwas Zusatzaufwand
Median von Medians	Beste theoretische Wahl	Hoher Rechenaufwand

Median: A[l], A[mid], A[r] ; Nehme Wert in Mitte

Aufwand: Worst Case N^2

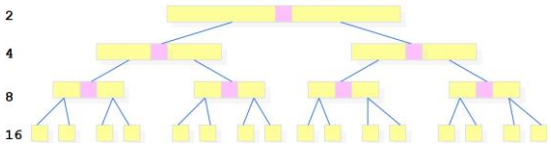
ein Bereich muss $\log_2(N)$ mal geteilt werden:

- Es entsteht dabei ein binärer Partitionenbaum mit Tiefe $\log_2(N)$.
- Der Aufwand, auf jeder Schicht diese komplett zu partitionieren, ist proportional zu N.

der Gesamtaufwand ist somit proportional zu $N \times \log_2(N)$.

Die Ordnung von Quicksort ist somit $O(N \times \log(N))$,

- wenn bei jeder Partitionierung eine gleichmässige Aufteilung der Daten erfolgt



Obere Schranken & Pruning

Schätzung: Eine obere Schranke $b(v)$ ist eine optimistische Vorhersage des besten möglichen Werts, der über einen Knoten v erreicht werden kann.

Optimismus: $b(v) \geq f(v)$, d. h., die Schranke überschätzt den tatsächlichen Wert.

Zweck: Ermöglicht das Abschneiden (Pruning) von Pfaden, deren Schranke unter einer bekannten Lösung liegt.

Priorisierung von Knoten:

- Die Zielfunktion $f(v)$ bewertet, wie gut eine aktuelle Lösung ist.
- Die obere Schranke $b(v)$ hilft zu entscheiden, ob es sich lohnt, diesen Pfad weiter zu verfolgen.

Pruning:

- Falls $b(v) <$ beste bisher gefundene Lösung, wird der Knoten v verworfen.

*Algorithmisches Beispiel: A-Algorithmus**:

- Zielfunktion: $f(v) =$ bisherige Wegkosten + geschätzte Restkosten.
- Schranke: Luftliniendistanz als Mindestwert der Restkosten.

Wahl Sortieren

Bibliotheksfunktion -> Collections.sort oder Arrays.sort

wenige Datensätze (weniger als 1000),

- Laufzeit unerheblich,
- möglichst einfachen Sortieralgorithmus wählen (also Insertion-Sort, Selection-Sort oder Bubble-Sort).

vorsortierte Datenbestände

- dann Insertion- oder Bubble-Sort .

viele ungeordnete Daten

- dann Quick-Sort bevorzugen.

viele Daten, ungeordnet, sehr oft zu sortieren

- Distribution-Sort an das spezielle Problem anzupassen

sehr viele Daten

- externes Sortierverfahren in Kombination mit schnellem internem

Optimierung durch Parallelisierung

Methode	Vorteil	Nachteil	Eignung
Naive Threads	Einfach zu implementieren	Hoher Overhead, ineffizient bei zu vielen Threads	Nicht empfohlen
Threshold für Parallelisierung	Reduziert Overhead	Muss optimiert werden	Gute Wahl für einfache Implementierungen
Thread-Pools	Effizientere Nutzung der Ressourcen	Komplexer als einfache Threads	Sehr gute Wahl für Multi-Core-CPUs
Fork/Join-Framework	Optimal für rekursive Algorithmen	Benötigt Einarbeitung	Beste Wahl für große Datenmengen

Nein Effizierter, aber Komplexität gleich

Suchen&Sortieren

Binary/Linear Search

Nach jeder suche wird Array halbiert, sortiert!

Algorithmus	Best-Case	Average-Case	Worst-Case
Lineare Suche	O(1)	O(n)	O(n)
Binäre Suche	O(1)	O(log n)	O(log n)

Intersect Search O(n+m)

Zwei Zeiger (Pointer) setzen:

- Setze `i = 0` für die erste Liste (`A`).
- Setze `j = 0` für die zweite Liste (`B`).

Vergleich der Elemente an Position `i` und `j`:

- Falls `A[i] == B[j]`:
  → Das Element ist in **beiden Listen enthalten**, also speichere es und **inkrementiere beide Zeiger** (`i++` **und** `j++`).
- Falls `A[i] < B[j]`:
  → `A[i]` ist kleiner, also erhöhe `i` (suche in `A` weiter).
- Falls `A[i] > B[j]`:
  → `B[j]` ist kleiner, also erhöhe `j` (suche in `B` weiter).
- Wiederhole diesen Prozess, bis eine der Listen zu Ende ist.

Hash Funktion

Eigenschaften:
- Deterministisch
- Schnell berechenbar
- Gleichmäßige Verteilung
- Minimal kollisionsanfällig
- equals=gleicher Code JAV

Index wird gehasht und damit Hashtabelle erstellen mit O(1) für Zugriff

Kollisionen:
- Seperate Chaining: Jeder Index ist eine Liste
- Open Adressing (LF < ~0.8): Linear Probing und Quadratic Probing für freien Platz

L/Q Probing

Normalerweise O(1), aber bricht ein bei hohem LF oder ungünstiger Verteilung. **Bei Quadratic**: ^2

String Suche

Bruteforce: O(n*m)

KMP: O(n+m) wird nicht verwendet, BF doppelt so schnell

Fuzzy Search

Levenshtein Distanz: Anhand nötiger Buchstabenänderungen

Trigram: Anhand maximaler Anzahl gleicher Wortgruppen „Peter“, die 3-er Gruppen „PET“, “ETE“, “TER“.

Phonetische Suche: 1. Buchstabe + 3 Ziffern. Tabelle von Buchstaben= Ziffer, mit 0 sonst

Bubble Sort

Wiederholtes Tauschen von benachbarten Elementen

Optimierung: Abbruch, wenn keine Vertauschung erfolgt

Best Case	*Average Case*	*Worst Case*
O(N)	$O(N^2)$	$O(N^2)$

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}$$

Invertiertes Dateisystem

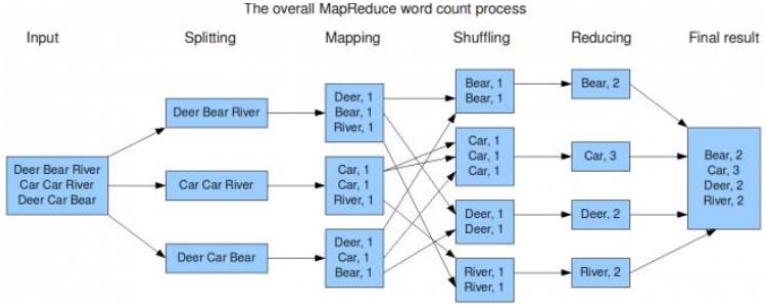
Direkten Dateien: Originaldokumente.

Index (Wortliste): alphabetische Liste aller Wörter.

Invertierte Dateien: Referenzen zu den Dokumenten, in denen das Wort vorkommt, sowie Zusatzinformationen (z. B. Position, Häufigkeit).

Vorteile: Schnelle Suche, Effizienz, Erweiterbarkeit

Map Reduce / parallele Streams



Java mit Streams: Fork, Compute, Join **ABER** Ordnung geht verloren. Performance kann auch schlechter werden durch Overhead. Auch Probleme Wegen Threads können erstehen.

Selection Sort

Unterteilung auf Sortiere und nicht sortierter teil. Kleinstes Element an Ende platzieren.

Vorteil: Weniger Swaps

Nachteil: Immer $O(n^2)$

Insertion Sort

Wie Selection Sort aber, ein random Element wird an die richtige Stelle sortiert. Weniger Vergleiche aber mehr Swaps. Daher gut bei kurzen Datensätzen und vorsortieren.

Stabilität

Wird Reihenfolge von gleichen Schlüssel bewahrt?

Stabil: Bubble Sort, Insertion Sort, Merge Sort

Instabil: Selection Sort, Quick Sort

Distribution Sort

Element wird direkt auf korrekten Platz eingefügt. Z.b. PLZ. **Vorteile**: O(n)

Nachteile: Plätze müssen bekannt

Hashtabellen

Vorteile
- Suchen Einfügen in Hash-Tabellen sehr effizient
- "Einfache" Binären Bäumen können bei ungünstigen Inputdaten degenerieren, Hash-Tabellen kaum.
- Der Implementationsaufwand für Hash-Tabellen ist geringer als derjenige für ausgeglichene binäre Bäume.

Nachteile
- das kleinste oder grösste Element lässt sich nicht einfach finden
- Geordnete Ausgabe nicht möglich
- die Suche nach Werten in einem bestimmten Bereich oder das Finden z.B. eines Strings, wenn nur der Anfang bekannt ist, ist nicht möglich

Hash-Tabellen sind geeignet wenn: die **Reihenfolge** nicht von Bedeutung ist nicht nach **Bereichen** gesucht werden muss die **ungefähre (maximale) Anzahl** bekannt ist.

Bei Löschen wegen Probing Probleme: Als gelöscht Markieren oder rehashing (Wegen Probing = Lücke)

Extendible Hashing: Wenn ein Bucket überläuft (d. h., er hat zu viele Elemente), wird nicht die gesamte Hash-Tabelle erweitert, sondern nur der betroffene Bucket gesplittet.

Global Depth (GD): n von rechts Bits oder mod 2^n

Local Depth (LD): Anzahl der Bits, die für einen bestimmten Bucket gelten.

Hinzufügen: Wenn GD = LD => GD+1, sonst Bucket aufteilen LD+1 bei beiden neuen.

GD+1: Verdoppeln, bei neuen 1 am Anfang. Nicht direkt gebrauchte Buckets verlinken auf alte

Externes Sortiern

Phase 1: Sortieren-Verteilen
- Lade jeweils einen Teil der zu sortierenden Datei in den Speicher.
  - Sortiere diese
  - Schreibe die sortierten einzelnen Daten in mind. 2 Ausgabedateien
- ⇨ Es entstehen Folgen von sortierten Abschnitten

Phase 2: Mischen
- Lese von den Ausgabedateien das erste Element
- Schreibe das kleinere und lese das nächste Symbol der gleichen Datei. Länge der geordneten Abschnitte verdoppelt sich.

Aufwand: (n Sequenzen, m Eingabedateien)
- Anzahl Mischphasen: O(logm(n))
- Mischen: O(n * logm(n))

Speicher

Statisch Zuteilung

Vorteile:
- beim Start bekannt, ob Speicher ausreicht & Einfach

Nachteile:
- die Grössen aller Datenstrukturen müssen zur Übersetzungszeit bekannt sein.
- es muss so viel Speicher angefordert werden, wie das Programm maximal benötigt
- keine rekursiven Aufrufe möglich.
- keine dynamischen Datenstrukturen wie Listen und Bäume möglich.

Stack Zuteilung

Vorteile:
- rekursive Aufrufe sind möglich.
- effiziente Zuteilung/Freigabe von Speicher (SP, FP).

Nachteile:
- Grösse der einzelnen Datenstrukturen muss bekannt sein.
- Der Aufrufer kann nicht auf die Werte des Aufgerufenen nach dessen Rückkehr zugreifen.
- D.h. die Werte auf dem Stack sind nach dem Verlassen der Methode verloren.
- Stack-Overflow möglich.

Heap Zuteilung

Vorteile:
- rekursive Aufrufe sind möglich.
- die Grösse von Datenstrukturen kann zur Laufzeit festgelegt werden.
- dynamische Datenstrukturen

Nachteile:
- Zuteilung/Freigabe der Daten ist rechenintensiv und kompliziert.
- Speicher muss explizit vom Programm angefordert und wieder freigegeben werden -> Programmierfehler möglich.
- Heap-Overflow möglich.

Speicherverwalter

```
class Storage {
    long malloc (int size),
    void free(long addr);
}
```

Gibt Adresse zurück. Hat 2 Listen: Belegt, Frei Bei objektorientierten Sprachen unpraktisch, die Grösse ist dem System bekannt also Grössengabe redundant

Anfordern:
1. Block wird von Frei -> Belegt
2. Rest des Bereichs (Verschnitt) wird in die Frei-Liste eingetragen.
3. Referenz auf den Bereich wird zurückgegeben.

Freigeben: der Speicher wird aus der Belegt-Liste entfernt und in die Frei-Liste eingetragen

Fragmentierung: Es bilden sich mit der Zeit Löcher im Speicher. Als Lösung wird periodisch kompaktiert.

Fehler bei Heap:

Vergessen den Speicher anzufordern => Zufälliger Wert/ Bereits benutzter Platz ➔ Mit null initialisiere

Zuwenig Speicher angefordert => Nachbar überschrieben ➔ Überprüfen ob Zugriff im gültigen Bereich ggf. erweitern

Vergessen den Speicher freizugeben: Memory Leak => das Programm benötigt immer mehr Speicher ➔Speicherverwalter muss freigeben/ Garbage Collection

Der Speicher wird freigegeben, obwohl er noch verwendet, wird: Dangling Pointer => wenn der Speicherbereich wieder neu vergeben wird, dann zeigt die alte Variable immer in diesen Bereich. Es wird eine anderweitig benutzte Speicherstelle zugegriffen und verändert ➔ der Speicherverwalter gibt den Speicher automatisch frei (Java)

Automatische Speicherverwaltung

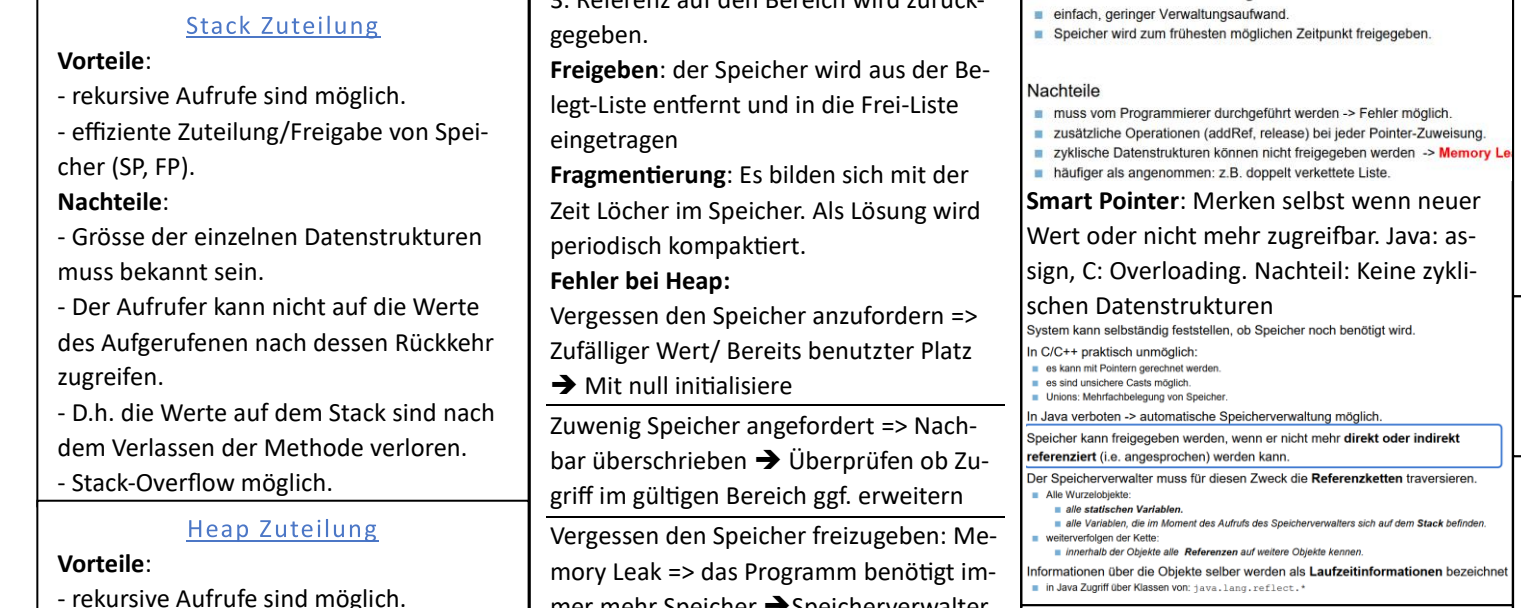
Hauptaufgabe: Freigabe des nicht mehr benötigten Speichers

Einfache Verfahren ohne Laufzeit Info: Referenzzählung &/ Smart-Pointer

Garbage Collector: Mark-Sweep GC &/ Copying GC &/ Generational GC

Referenzzählung

Es wird gezählt wie viele Referenzen auf ein Objekt verweisen, wenn 0 dann löschen.



Smart Pointer: Merken selbst wenn neuer Wert oder nicht mehr zugreifbar. Java: assign, C: Overloading. Nachteil: Keine zyklischen Datenstrukturen

Weak References

Objekte im Speicher halten, welche trotzdem GC werden können.

Verwendung: Caches, Listener, Observable, WeakHashMap

Mark-Sweep GC

Speicher wird bei Bedarf frei gegeben

Mark: Von Wurzel alle erreichbaren Blöcke markieren

Sweep: sequenziell nicht markierte freigeben und Markierungen löschen

Vorteile: keine zusätzlichen Operationen bei Pointer-Zuweisungen nötig. zyklische Datenstrukturen können aufgelöst werden.

Nachteil: +Aufwand, Programm wird gestoppt "Stop the World" GC genannt, Fragmentierung

Copying GC

Der Heap-Speicher wird in zwei gleich große Teile geteilt: **From-Space** (aktuell genutzter Speicher) und **To-Space** (freier Speicher).

Wenn From-Space voll: Markieren => Aktiv kopieren zu To => Umschalten (From ist komplett frei, To ist aktiv)

Vorteile: Fragmentierungsfrei, Konstante Allokationsgeschwindigkeit, Effiziente Bereinigung

Nachteile: Doppelter Speicher, Performance, Stop

Cheney's Copying Algorithm: Breitensuche, Scan-Pointer → Nächstes zu überprüfen, Free-Pointer → Nächstes freie

Generational GC

Schwach: most objects die young, Stark: older => less

Wenn New Gen voll ist, lebend zu Old, Rest gelöscht. New Gen meistens Mit Copying GC, old mit Mark-Sweep

Finalizer

Ein **Finalizer** ist eine Methode (protected void finalize() in Java), die aufgerufen wird, bevor ein Objekt von der Garbage Collection (GC) entfernt wird.

Wird genutzt, um Ressourcen wie Dateien, Netzwerkverbindungen oder Datenbankverbindungen zu bereinigen.

Probleme: Unvorhersehbare Ausführung, Performance, Sicherheitsrisiken

Lösung: AutoCloseable & try-with-resources: Methode wird explizit aufgerufen, **sofort wenn das Objekt nicht mehr benötigt wird**.