ADS Summary (mündlich, beliebig lang)

26. September, 2023; rev. 16. Januar 2024

Linda Riesen (rieselin)

1 Vorlesung 01

Ein Algorithmus ist eine Anleitung zur Lösung einer Aufgabenstellung, die so präzise formuliert ist, dass sie "mechanisch" ausgeführt werden kann.

Eigenschaften v. Algorithmen

- Determiniertheit: Identische Eingaben führen stets zu identischen Ergebnissen.
- Determinismus: Ablauf des Verfahrens ist an jedem Punkt fest vorgeschrieben (keine Wahlfreiheit).
- Terminierung: Für jede Eingabe liegt das Ergebnis nach endlich vielen Schritten vor.
- Effizienz: «Wirtschaftlichkeit» des Aufwands relativ zu einem vorgegebenen Massstab (z.B. Laufzeit, Speicherplatzverbrauch).

1.1 Abstrakte Datentypen (ADT's)

Bsp Java: Interfaces, Klassen

- Information Hiding: (nur was absolut nötig public)
- Schnittstelle: von aussen sichtbar, Zugriffsmethoden stellen sicher das innere Logik unverändert bleibt
- Enthält Beschreibung, was die Operationen tun
- Implementation: kann verändert werdne ohne dass von aussen bemerkt, gibt versch. Implementationsmöglichkeiten

1.1.1 Stack (= Stapel / Kellerspeicher): als ADT

- neue Objekte können nur oben auf den Stapel gelegt werden
- auch das Entfernen von Objekten vom Stapel ist nur oben möglich
- push, pop, isEmpty, peek (oberst. Element ohne entfernen), removeAll, isFull

Anwendungsbeispiele Klammermatcher, XML Datei Prüfen, Parsen von Programmiersprachen (zbsp Rechnung)

1.1.2 Liste als ADT

- Grundlegenede Datenstruktur d. Informatik
- mit Liste kann Stack variabler Grösse implementiert werden
- add, add(pos), get, remove, size, isEmpty
- Eingesetzt wenn:
  - Anzahl der Elemente zur Erstellungszeit unbekannt (sonst meist Array)
  - Reihenfolge/Position ist relevant
  - Einfügen und Löschen von Elementen ist unterstützt
  - Implementierung v. Stack/Queue
  - Bsp: Disk-Blöcke, Prozesse, Threads etc.

Struktur d. Listenknotens (Linked List) Enthält: Daten, Referenz auf Nächste Liste
Bei der Liste muss jeweils (im Schnitt) die Halbe Liste durchlaufen werden zur Bestimmung des i-ten Elements -> in Java gibts dafür den Iterator der sich die Positionen speichert
Wenn head = null, Liste ist leer

- teuere, lange Laufzeit: letztes Element löschen
- kurze Laufzeit: neues Element am Anfang/2. Position einfügen/ löschen

Doppelt Verkettete Liste kann mit next() in beide Richtungen arbeiten (Verkettet von Head und Tail)

Zirkulär Doppelt Verkettete Liste Letztes Element der Liste zeigt auf den Anfang

Sortierte Listen einzig anders: insert() fügt Elemente sortiert in Liste ein (Comparable/Comparator)

Array List bei einfügen am Anfang/Mitte müssen alle Nachfolgenden Elemente in Array umkopiert werden, schnell beim Zugriff auf bestimmtes Element, langsam bei Einfügen und Löschen
Langsamer als LinkedList für einfügen an xx Ort

1.1.3 Queue als ADT

- Eine Queue speichert Objekte in einer (Warte-)Schlange. Sie werden in der selben Reihenfolge entfernt, wie sie eingefügt werden (**FIFO**: First In First Out).
- enqueue (einfügen v. Objekt), dequeue (entf. und ältestes Obj. zurückgeben), peek, removeAll, isEmpty, isFull
- poll (Entfernung erstes Objekt)
- Anwendung: Warteschlange (bsp Drucker), Zugriff nicht parallel s. gestaffelt

Priority Queue enqueue hat zusätzlich priority die mitgegeben muss

2 Vorlesung 02

2.1 O-Notation

Zeitkomplexität & Speicherkomplexität
Nicht absolute Zahlen, sondern als Funktion von Grösse oder Anzahl Werten: n

- Laufzeit eines Algorithmus ist besonders für grosse Eingaben interessant
- Nicht die exakte Laufzeit interessiert, sondern die Grössenordnung
  - O(1) konstanter Aufwand (Bsp. Konstante)
  - O(log n) logarithmischer Aufwand
  - O(n) linearer Aufwand (Bsp. Schleife, Suche in dp verketteter Liste)
  - O(n·log n) linear-logarithmischer A.
  - O(n^2) quadratischer Aufwand (Bsp. Geschachtelte Schleifen)
  - O(n^k), k>1 polynomialer Aufwand
  - O(k^n) exponentieller Aufwand
  - O(n!) faktorieller Aufwand
- Rechenregeln
  - O(f*g) = O(f)*O(g)
  - O(r*f) = O(f) [mit r = Multiplikator]
  - O($\log_r(s*n)$) = O(log(n))
  - O(r^{s*n}) = O(rs*n)
  - O(f+g) = O(f) [mit O(f)>O(g)]
  - O(n^r) + O(n^s) = O(ns) [mit s>r>0]
  - O($r^n + n^s$) O(rn) [mit r>1]

2.2 Listen (siehe 01)

3 Vorlesung 03

3.1 Erweiterte Konstrukte (Fortsetzung ADT's)

3.1.1 Comparable

Um in einer Methodensignatur eine Klasse zu erlauben, die das Comparable Interface implementiert, müssen Sie folgende Deklaration verwenden: <? extends Comparable>

3.1.2 Type Erasure

zur Laufzeit steht keine Typeninformation zur Verfügung, der Typ vom Compiler gelöscht wird

3.1.3 Sets

- Ungeordnete Menge ohne Duplikate
- im Vergl. zur Liste fehlt alles Index bezogene (get, add(index), remove)
- Java: HashSet, TreeSet (alphabetisch)

Mengenoperationen containsAll (= Teilmenge), addAll (= Union), retainAll (= Intersection), removeAll (= Difference)

3.1.4 Wert u. Referenztypen

Referenz Bsp: Array, Objekt(-Klassen), String, ... = Zeiger auf Element sobald implementiert
Wert Bsp: int
Autounboxing (Interger <=> int direkt)

3.1.5 Generics

Collection Klassen spezialisiert auf mitgegebenen Elementtyp (Parametrisierter Datentyp, Typparameter)

Vorteile Java Generics

- Erhöhung Effizienz, Lesbarkeit, Aussagekraft, erleichtertes Verständnis, Performance (nicht aber weniger Speicherplatz)
- Weniger Fehler zur Laufzeit, mehr bereits zur Compile Zeit angezeigt
- Cast entfällt
- Felstlegung d Typen zur Übersetzungszeit

Generizität ohne Typparameter

- Überladen von Methoden (gleicher Name, andere Parameter/Rückgabewert)
- Object als Parameter
- Für jdn Datentypen eine eigene Klasse (IntBox, StringBox ...)

Gererizität mit Typparametern Collection von Beliebigen Typen: Collection<?>
Platzhalter für den Typ:

- E: Element (Collections)
- K: Key
- N: Number
- T: Type
- V: Value
- S, U, V: etc 2nd, 3rd, 4th Types
- ?: Wildcard ("Vergessen von Typinformation", Ermöglichen von zuweisen von Unterschiedlichen Typen, Zsmführen von Unterklassen) [nur für lokale Variablen und Methoden Parameter]
- Bounded Wildcard: Bsp: List<? extends Figure> / List<? super Rectangle>

Regeln dabei:

- Wenn Foo Subklasse von Bar, dann ist G<Foo> keine Subklasse von G<Bar>
- List<String> kann nicht List<Object> zugewiesen werden

4 Vorlesung 04

Rekursion Ein Algorithmus/Datenstruktur heisst rekursiv definiert, wenn er/sie sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist.
Dadurch kann eine unendliche Menge durch endliche Aussage beschrieben werden, entspricht der Vollständigen Induktion (gibt einen Basis Fall (Startbedingung) und einen Allgemeinen Fall (Induktionsschritt, Problem wird bis xy Fortgesetzt (bis Basis Fall erreicht wird))
Jeder Rekursive Alg. hat äquivalenten iterativen Alg.!!!

4.1 Vor/Nachteile Rekursion

kürzere Formulierung, leicht Verständliche Lösung, Einsparen v. Variablen, teilweise s. effizient
weniger effizient Laufzeitverhalten (Overhead bei Methodenaufruf), braucht mehr Speicher (Rücksprungadressen, lokale Variablen)

4.2 Rekursionsarten

4.2.1 Indirekte / Direkte Rekursion

Direkt: Methode ruft sich selbst wieder auf
Indirekt: 2 Methoden rufen sich gegenseitig auf (häufige Fehlerquelle)

4.2.2 Endrekursion (tail rekursion)

Ein Programm bei dem der rekursive Aufruf die allerletzte Aktion in jdm Zweig ist, lässt sich meist einfach in iterative Form überführen (iterativ ist effizienter, da Stackverwaltung bei rekursiven Alg. ineffizient)

4.3 Rekursionstiefe, Speicherkomplexität, Zeitkomplexität

Rekursionstiefe: Maximale Tiefe"der Aufrufe einer Methode minus 1
Zeitkomplexität: Rechenaufwand: O(x) Verdoppelung in jdm Schritt
Speicherkomplexität: Wächst mit O(x) -> wann wird Speicher freigegeben

5 Vorlesung 05

5.1 Bäume in der Informatik

Beispiele: XML Dokument, Dateisystem, Ausdruckbaum (a+b*c)
Definition: ein Baum ist leeröder er besteht aus einem Knoten mit 0, 1 oder many disjunkten Teilbäumen [Baum = leer, Baum = Knoten(Baum)*]

- Alle Knoten ausser der Wurzel (root) sind Nachfolger (descendant, child) genau eines Vorgänger-Knotens (ancestor, parent).
- Knoten mit Nachfolger werden als innere Knoten bezeichnet
- Knoten ohne Nachfolger sind Blattknoten.
- Knoten mit dem gleichen Vorgänger-Knoten sind Geschwisterknoten (sibling).
- Es gibt genau einen Pfad vom Wurzel-Knoten zu jedem anderen Knoten.
- Die Anzahl der Kanten, denen wir folgen müssen, ist die Weglänge (path length).
- Die Höhe (oder Tiefe) eines Baumes gibt an, wie viele «Ebenen» der Baum hat: Anzahl Kanten + 1.
- Das Gewicht (oder Grösse) ist die Anzahl der Knoten des (Teil-)Baumes
- Das (ev. rekursive) Besuchen aller Knoten im Baum wird als durchlaufen/traversieren bezeichnen

Implementation d. TreeNodes, jd Knoten zeigt auf Nachfolgerknoten

5.1.1 Binärbaum

ein Knoten hat max 2 Nachfolger [Baum = leer, Baum = Knoten (Baum Baum)]
Ein AVL Baum ist ein ausgeglichener binärer Suchbaum, wobei sich für jeden Knoten die Höhen seines linken und rechten Teilbaumes um maximal eins unterscheiden

- B-Baum ist beste Möglichkeit f Speicherung auf Festplate

- Tiefe/Höhe: Anzahl Ebenen
- auf jeder Höhe h: max. $2^{(h-1)}$ Knoten
- Maximale Anzahl Knoten: $2^h - 1$
- In einem binären Suchbaum hat das grösste Element keinen rechten Nachfolger.
- Binärbaum heisst voll (full) wenn jd Knoten entweder Blatt oder 2 Kinder
- Binärbaum heisst vollständig (complete) wenn alle Ebenen (bis auf die letzte) voll gefüllt und Blätter i. d. letzen Ebene linksbündig angeordnet
- Mögliche Traversierungsarten: (n = Knoten, A, B Kinder)
  - Preorder: n, A, B
  - Inorder: A, n, B (keine Queue verwenden)
  - Postorder: A, B, n
  - Levelorder: n, a0, b0, a1, a2, b1, b2

5.1.2 Traversierung Visitor Pattern

Wie kann ich den Baum traversieren (z.B. in Preorder) und während der Traversierung verschiedene Aktionen je Knoten ausführen, ohne dass ich die Klasse der Traversierung oder die Klasse des Baums erweitern muss?
-> Wir lagern die Methoden zum Verarbeiten der Knotendaten in eine eigene Klasse (die Visitor-Klasse) aus und rufen während der Traversierung diese jeweils auf.

5.1.3 Methoden zur Mutation v. (sortierten) Bäumen

Einfügen unsortiert einfache rekursive Methode (insertAt(T x)) die neues Element an Schluss einer Liste anhängt (Baum wird als erweiterte Liste angeschaut)

Einfügen sortiert (Beim binären Suchbaum (= Sortierter Baum) werden Objekte anhand Schlüsselwerts geordnet eingefügt) Für jeden Knoten gilt:

- im linken Unterbaum sind alle kleineren Elemente KL <=* k
- im rechten Unterbaum sind alle grösseren Elemente: KR >* k

Suchen rekursiv (sortiert) Bei einem vollständigen (resp. kompletten) Binärbaum müssen lediglich Log2 Schritte durchgeführt werden bis Element gefunden wird.
Entspricht dem Aufwand des binären Suchens. (s. effizient)

Löschen Sortiert

1. den zu entfernenden Knoten suchen
2. Knoten löschen. Dabei gibt es 3 Fälle:
   - Fall: der zu löschende Knoten hat keinen Teilbaum => Knoten löschen
   - Fall: der Knoten hat genau einen Teilbaum => Knoten löschen und Referenz neu setzen
   - Fall: der Knoten hat zwei Teilbäume => Es muss ein Ersatzknoten oder Ersatzwert gefunden werden (man kann den Knoten austauschen, oder den Inhalt des zu löschenden Knotens ersetzen).

6 Vorlesung 06

6.1 Suchen und Tiefe

6.1.1 Suche x im Baum:

- Wenn x == Wurzelelement gilt, haben wir x gefunden.
- Wenn x > Wurzelelement gilt, wird die Suche im rechten Teilbaum fortgesetzt, sonst im linken Teilbaum.
- Bei vollständigem Binärbaum müssen Log2(n) Schritte durchgeführt werden

6.1.2 Zugriffszeiten und Tiefe

Die Zugriffszeit (Suchen, Einfügen und Löschen) von Elementen ist proportional zur Höhe / Tiefe des Baumes.

- Ziel: bei gegebener Anzahl Elemente ein Baum mit möglichst geringer Tiefe.
- Probleme:
  - neue Knoten können nur unten angehängt werden
  - einzufügende Elemente sind meist nicht à priori bekannt
  - bei «unglücklicher» Reihenfolge entstehen sehr ungleichmässige, d.h. «unbalancierte» Bäume

6.2 Balanciertheit von Suchbäumen

Wenn man Daten in beliebiger Reihenfolge in einen Binärbaum «naiv» (wie bisher «einfach so») einfügt, werden die beiden Teilbäume vermutlich unterschiedlich schwer und unterschiedlich tief sein.
Schlimmstens: Daten werden in Sortierter Reihenfolge eingefügt, Baum wird zur Liste, Aufwand wird zu O(n)

6.2.1 Vollständig balancierter Baum (vollständig ausgeglichener Baum)

Ist ein Baum, bei dem, abgesehen von der untersten Ebene, alle Ebenen vollständig (mit Knoten) besetzt sind.

- Tiefe beträgt: log2(n+1)
- Dies ist die ideale Tiefe für binäre Suchbäume, der zeitliche Aufwand zum Suchen ist optimal: O(log2 (n)).
- Aber: beim Einfügen, Löschen und Ändern muss der Baum möglicherweise vollständig reorganisiert werden: O(n) – es gibt bessere Lösungen.

6.2.2 Balancierter Baum

Ist ein Baum, der eine maximale Höhe von c1·log(n) + c2 garantiert

- Maximale Höhe: c1·log(n) + c2 mit [c1 und c2 sind Konstanten]
- Aufwand der Suche noch immer O(log(n)).
- Es sind unterschiedliche Regeln möglich. Die Regeln können sich z.B. auf Höhen, Gewicht oder Struktur der Bäume beziehen.
- Es gibt viele verschiedene Baumarten und Algorithmen, die dieses Kriterium der maximalen Höhe erfüllen.

6.2.3 AVL-Baum

Der AVL-Baum ist ein balancierter Baum, bei dem für jeden Knoten gilt, dass sich die Höhe der beiden Teilbäume um höchstens eins unterscheidet.

- Maximale Tiefe: c1 ·log2 (n+2) + c2 mit [c1 ≈ 1.44, c2 ≈ -0.33]
- Etwa 44% höher als ein vollständig ausgeglichener Baum.
- Beim Einfügen und Löschen sorgt man dafür, dass die AVL-Ausgleichbedingung erhalten bleibt. (Es muss Buch geführt werden, wie tief die darunter gelegenen Teilbäume sind (pro Knoten eine Zahl), Wird die Differenz zwischen linkem und rechten Teilbaum grösser 1 muss etwas unternommen werden.)
- O(log(n))
- Suchen und Traversieren unverändert (ist und bleibt ein Binärbaum)
- Zum Wiederherstellen der Ausgleichsbedingung werden sogenannte Rotationen eingesetzt.
- Eine Höhendifferenz von 3 ist nicht möglich, da nach jedem Einfügen/Löschen ausbalanciert wird.

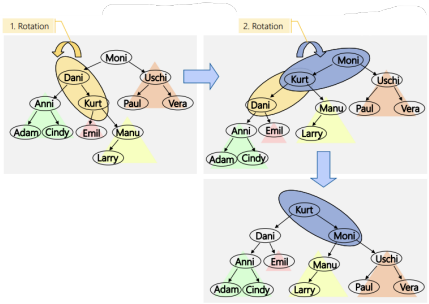

Abbildung 1: Doppelrotation

6.2.4 B-Baum

Ein B-Baum ist ein vollständig balancierter Baum. In der Ordnung n (n = max. Anzahl Kinder) enthält jeder Knoten, ausser der Wurzel, mindestens $\lfloor (n-1)/2 \rfloor$ und höchstens n-1 Schlüssel. Alle Blätter haben die gleiche Tiefe.

- Die Wurzel hat 1 bis n-1 Schlüssel.
- Tiefe des Baumes $\approx log_{AnzahlVerweise}(AnzahlElemente)$
- Für die Schlüssel und Verweise gilt:
  - innerhalb eines Knotens sind alle Schlüssel sortiert
  - alle Schlüssel im (i-1)-ten Nachfolgerknoten sind kleiner oder gleich dem Schlüssel s_i
  - alle Schlüssel im i-ten Nachfolgerknoten sind grösser als der Schlüssel s_i
- Die inneren Knoten enthalten Schlüssel und Verweise (auf Nachfolgeknoten und Informationen).
- Sind die Informationen nur in den Blättern gespeichert, spricht man vom $B^+ - Baum$.
- Benutzt für Dateisysteme (Massenspeicherzugriffe)
- Es gibt mindestens n/2 Unterbäume (n wird bewusst an die Grösse der Speicherseite angepasst). Dadurch wird der Baum weniger hoch => weniger Plattenzugriffe notwendig.

B-Baum Einfügen/Löschen Eingefügt wird immer in den Blättern (solange Platz).
Ist dieser bereits voll, dann gibt es einen Überlauf / Bereits Leer: Unterlauf

- Aufteilen in zwei Knoten und «heraufziehen» des mittleren Elements in den Vaterknoten.
- Falls der Vaterknoten überläuft: Vorgang wiederholen. (Kann bis zur Wurzel propagieren, was dann die Höhe des Baumes ändert)

B-Baum: Suchen Den Wurzelblock lesen, Gegebenen Schlüssel S auf dem gelesenen Block suchen, Wenn gefunden => referenzierte Daten lesen => fertig
Ansonsten i finden, sodass: Si <= S < Si+1, Block des i-ten Nachfolgeknoten einlesen, Schritte 2 bis 5 wiederholen

6.2.5 2-3-4 Baum, Rot-Schwarz Baum

Ein 2-3-4-Baum ist ein Spezialfall des B-Baums (mit Ordnung 4), in dem jeder Knoten zwei, drei oder maximal vier Kinder besitzt.
Ein Rot-Schwarz-Baum ist ein Spezialfall des 2-3-4-Baums, bei welchem die Knoten mit 2 oder mehr Schlüsseln durch Binärbäume implementiert werden: Durch «Färben» der Kanten wird 2-3-4-Baum als Binärbaum implementiert.
Ausgleichverfahren so dass Rot-Schwarz-Bedingungen erhalten bleiben:

- Jeder Knoten im Baum ist entweder rot oder schwarz.
- Die Wurzel des Baums ist schwarz.
- Null-Zeiger (früher NIL statt Null) für fehlendes Kind, betrachten wir als externe Knoten mit der Farbe schwarz.
- Kein roter Knoten hat ein rotes Kind.
- Jeder Pfad, von einem gegebenen Knoten zu seinen Blattknoten, enthält die gleiche Anzahl schwarzer Knoten (Schwarzhöhe/Schwarztiefe). Für die Tiefe zählt man nur die Schwarzen Knoten.

7 Vorlesung 07

7.1 Typische Fragestellungen Graphen

- Kürzeste Verbindung (Shortest Path)
- Maximaler Durchsatz (Maximal Flow): Verkehr
- Kürzester Weg f alle Punkte (Traveling Salesman)
- Reihenfolge (Topological Sort): von Tätigkeiten aus Netzplan
- Minimal benötigte Zeit (Critical Path): optimale Reihenfolge

7.2 Definitionen

Graph G = (V, E) besteht aus einer endlichen Menge von Knoten V und einer Menge von Kanten E $\subseteq$ V x V.
Knoten Objekte mit Namen und anderen Attributen. (Vertex/Node)
Kanten Gerichtete Verbindungen zwischen zwei Knoten, allenfalls mit Attributen. Es können auch mehrere Kanten zwischen zwei Knoten bestehen und Kanten können einen Knoten mit sich selbst verbinden (Edge)
Pfad Sequenz von benachbarten Knoten, **einfacher Pfad** falls kein Knoten 2x Vorkommt, **zyklischer Pfad/geschlossener Pfad** falls Anfangs = Endknoten (kann beim Bäumen nicht sein)
Pfadlänge Anz Kanten des Pfades
2 Knoten sind **benachbart (adjacent, adjazent)** falls es eine Kante gibt die diese direkt verbindet
Vollständiger/Kompletter Graph Jeder Knoten ist mit jdm anderen Knoten direkt verbunden
Verbundener Graph Jeder Knoten ist mit mind. 1 anderen Knoten verbunden (= es existiert ein Pfad)
Dichte des Graphen Anzahl Knoten / Anzahl Kanten (zw 0 und 1)
Dichter/Dünner (= lichter) Graph nur wenige Kanten fehlen/sind vorhanden
Gerichtet/Ungerichteter Graph ungerichtet geht jd Kante in beide Richtungen, gerichtet nur in eine (Pfeile notwendig); in beiden können Zyklen sein; Einen ungerichteten Graphen kann man auch als gerichteten Graphen darstellen
gewichteter Graph (gewichtet + gerichtet = Netzwerk) Ein Graph G = (V, E) kann zu einem gewichteten Graphen G = (V, E, gw(E)) erweitert werden, wenn man eine Gewichtsfunktion hinzunimmt, die jeder Kante ein Gewicht gw(e) zuordnet Gewichtete Graphen haben **Gewichte (=Kosten)** und die **gewichtete Pfadlänge** ist Summe der Gewichte auf Pfad
Azyklischer Graph Ein gerichteter Graph ohne Zyklen wird als gerichteter azyklischer Graph (DAG) bezeichnet, aber er ist nicht notwendigerweise ein Baum.

7.2.1 Spezialfälle von Graphen

Baum gerichteter, zyklenfreier, verbundener Graph (jd Knoten hat genau 1 eingehende Verbindung, ein Knoten hat keine eingehenden Kanten (Wurzel)
Wald Gruppe von nicht zusammenhängenden Bäumen

7.3 Graph Implementationen

7.3.1 Adjazenz Liste

Jeder Knoten führt eine Adjazenz-Liste, welche alle Kanten zu den benachbarten Knoten enthält. Dabei wird für jede Kante (Edge) ein Eintrag bestehend aus dem Zielknoten und weiteren Attributen (z.B. Gewicht) erstellt.

7.3.2 Adjazenz Matrix

N x N (N = Anz. Knoten), boolean Matrix. Dort true, wo Verbindung existiert, falls gewichtet mit double statt boolean (mit 0/undefined für nicht verbunden)
Falls ungerichtet ist Matix symmetrisch zur diagonalen
Grosser Speicher Overhead, effizient und einfach zu implementieren, Speichereffizient falls dichter Graph

7.4 Graph Algorithmen

7.4.1 Traversierungen (Tiefensuche / Breitensuche

Tiefensuche (depth-first): Ausgehend von einem Startknoten geht man vorwärts (tiefer) zu einem neuen unbesuchten Knoten, solange einer vorhanden (d.h. erreichbar) ist. Hat es keine weiteren (unbesuchten) Knoten mehr, geht

man schrittweise rückwärts und betrachtet die noch unbesuchten Knoten. Entspricht der PreorderTraversierung bei Bäumen.
Breitensuche (breadth-first): Ausgehend von einem Startknoten betrachtet man zuerst alle benachbarten Knoten (d.h. auf dem gleichen Level), bevor man einen Schritt weitergeht. Entspricht der Levelorder-Traversierung bei Bäumen.

7.4.2 Kürzester ungewichteter/gewichteter Pfad

ungewichtet: Vom Startpunkt ausgehend werden die Knoten mit ihrer Distanz markiert: Der Graph wird mit Breitensuche traversiert. Eine neue Distanz wird nur eingetragen, wenn sie kleiner als der bestehende Distanzeintrag ist. Gleichzeitig wird noch eingetragen, von welchem Knoten aus der Knoten erreicht wurde. Vom Endpunkt aus kann dann rückwärts der kürzeste Pfad gebildet werden.
gewichtet: Algorithmus gleich wie vorher, aber korrigiere Einträge für Distanzen Der Eintrag für Knoten wird auf den neuen Wert gesetzt; statt «markiert» gehe so lange weiter, bis der neue Weg länger als der angetroffene ist. (Optimiert mit Dijkstra Alg.)

Dijkstras Algorithmus Aufwand $O(n^2)$
Teilt die Knoten in 3 Gruppen auf:

- besuchte Knoten (kleinste Distanz bekannt)
- benachbart zu allen bereits besuchten Knoten
- unbesehene Knoten (der Rest)

Solange nicht alle Knoten besucht wurden (grün sind):

1. Berechne für alle benachbarten, unbesuchten Knoten des aktuell besuchten Knotens (current) die neuen Gewichte (rote Pfeile).
2. Suche unter allen benachbarten (nicht nur jene des aktuellen Knotens), unbesuchten Knoten denjenigen, dessen Pfad zum Startknoten das kleinste Gewicht (= kürzeste Distanz) hat (grüner Rahmen um Zahl).
3. Besuche diesen (neuer aktueller Knoten und kürzester Pfad zu diesem Knoten bekannt).

7.4.3 Greedy (gierige) Algorithmen

Spezielle Klasse von Algorithmen, Sie zeichnen sich dadurch aus, dass sie einen Folgezustand auswählen, der zum Zeitpunkt der Wahl den grössten Gewinn bzw. das beste Ergebnis verspricht, berechnet durch eine (lokale) Bewertungsfunktion.
oft Schnell, können aber in lokalen minima/maxima stecken bleiben

7.4.4 Topologisches Sortieren

Die Knoten eines gerichteten, unzyklischen Graphs in einer «natürlichen» Reihenfolge (Sortierung) auflisten. In anderen Worten: Die Knoten eines gerichteten, azyklischen Graphen so in eine Reihe bringen, dass Pfeile immer nach rechts zeigen
Man zählt die Zahl der eingehenden Kanten von Knoten, die noch nicht aufgelistet wurden.

7.4.5 Maximaler Fluss

Die Kanten geben den maximalen Fluss zwischen den Knoten an. Was in einen Knoten hinein fliesst, muss auch wieder heraus. Es sind allenfalls mehrere Lösungen mit demselben Fluss möglich. (Zur Lösung: Original aufteilen in Vorläufiger Fluss und Restfluss, jeweils nicht benötigte (übrig) in Restfluss eintragen, solange weitermachen bis keine Verbindung mehr zw. Start und Ende in Restfluss sichtbar)

7.4.6 Traveling Salesman Problem (TSP)

Finden Sie die kürzeste Reiseroute, in der jede Stadt genau einmal besucht wird.
Option: am Schluss wieder am Ursprungsort
Ob es der kürzeste Weg ist, lässt sich nur durch Bestimmen sämtlicher möglicher Wege zeigen => O(n!).
Um schnell zu brauchbaren Lösungen zu kommen, sind meist durch Heuristiken (Die Kunst, mit begrenztem Wissen und wenig Zeit dennoch zu praktikablen Lösungen zu kommen) motivierte Näherungsverfahren notwendig, die aber in der Regel keine Güteabschätzung für die gefundenen Lösungen liefern.

Je nachdem, ob eine Heuristik eine neue Tour konstruiert oder ob sie versucht, eine bestehende Rundreise zu verbessern, wird sie als Eröffnungs- oder Verbesserungsverfahren bezeichnet.
Bis heute keine effiziente exakte Lösung bekannt
Näherungslösung mit O($n^2 * log(n^2)$): 1. Die Kanten werden nach ihren Kosten sortiert => Liste.
2. Wähle billigste Kante unter folgenden Bedingungen (ungültige Kanten ebenfalls aus Liste entfernen):

- Es darf kein Zyklus entstehen (eventuell am Ende erlauben, wenn Rundreise möglich)
- Kein Knoten darf mit mehr als zwei Kanten verbunden sein

8 Vorlesung 08

8.1 Trial and Error / Versuch und Irrtum (Versuch und Bewertung)

Älteste Lösungsstrategie überhaupt (Nat. Selektion), ziemlich rechen und zeitintensiv, erhält nicht unbedingt beste Lösung als Resultat (gibt auch Variante die einfach akzeptable Lösung sucht)

8.1.1 Backtracking: Entscheidungsbaum

Bsp Labyrinth: Gehe einen der Wege entlang... 1) bis zu einer Verzweigung => Rekursion 2) bis am Ziel => gefunden (Abbruch) 3) bis Sackgasse => dann gehe zur nächsten Verzweigung zurück die noch einen nicht probierten Weg aufweist (Rücksprung aus Rekursion)

Entscheidungsbaum Jd. Entscheidung entspricht einem Knoten, Teillösungen werden systematisch zu Gesamtlösungen erweitert bis Abbruch/Erweitern nicht mehr möglich, $O(z^n)$ mit [z = Verzweigungsgrad, n = max Tiefe]

Beispiele Springerproblem: Von einem beliebigen Schachfeld aus soll ein Springer nacheinander sämtliche Felder des Schachbretts genau einmal besuchen. (Idee: Mittels Backtracking alle möglichen Wege absuchen. = $O(8^{n*n})$)
Damenproblem: Es soll eine Stellung für acht Damen auf einem Schachbrett gefunden werden, so dass keine zwei Damen sich gegenseitig schlagen können. (Mittels Backtracking alle Kombinationen ausprobieren. = O(n!))
Rucksackproblem: Ein Dieb, der eine Wohnung ausraubt, findet K verschiedene Gegenstände unterschiedlicher Grösse und unterschiedlichen Werts, hat aber nur einen Rucksack der Grösse M zur Verfügung, um die Gegenstände zu tragen. (Idee: Mittels Backtracking alle möglichen Varianten ausprobieren = $O(2^n$ (mit n = anz Gegenstände))

8.1.2 Komplexität der Probleme: Erschöpfende Suche (exhaustive Search)

Es ist keine bessere Lösung bekannt als Ausprobieren aller Möglichkeiten (Trial and Error)
Der Aufwand bei diesen Algorithmen ist meist O(k^n) oder O(n!), entsprechend der Anzahl möglicher Kombinationen oder Permutationen
Dieser Aufwand wird aber schnell zu gross (Berechnung dauert zu lange)
Ausweg: jeder Teillösung wird eine "Güteßugeordnet

- Nichtdeterministisch polynomielle Zeitkomplexität: (z.B. Rucksackproblem mit O(2n))
- Polynomielle Zeitkomplexität: es gibt einen Algorithmus, der sich mindestens mit O(n^k) lösen lässt (z.B. Sortieren mit n·log(n)).

8.1.3 Zielfunktion

Umgehen der Kombinatorischen Explosion durch: Auswählen nur der Lösung die zum Ziel führt, Man berechnet zu jedem Knoten im Entscheidungsbaum den (besten) Zielwert, den man über diesen Knoten erreichen kann.
Bei der Zielfunktion wird meist eine obere (oder untere), geschätzt Schranke verwendet.

Algorithmus (schlecht) Für jeden Knoten:

- Berechne zu jedem Nachfolgeknoten im Entscheidungsbaum die Zielfunktion f(v).

- Gehe der Kante entlang (wähle die Teillösung aus), die zum Knoten mit dem höchsten Zielfunktionswert führt.

Problem gelöst:

- keine kombinatorische Explosion.
- sehr effizienter Algorithmus: $\sim$ Log(n)

Aber: Zur Berechnung der Zielfunktion muss das Problem meist schon gelöst sein, d.h. z.B. der Entscheidungsbaum unterhalb des Knotens vollständig durchlaufen sein.

Verbesserter Algorithmus Es wird nicht die exakte Zielfunktion, sondern eine einfacher zu bestimmende Funktion verwendet, ohne dass der gesamte Teilbaum untersucht werden muss: «Score der Lösung», Fitness Function, Kostenfunktion.
Es wird eine Bound-Funktion b bestimmt, die immer bessere Werte liefert als die exakte Zielfunktion f: obere Schranke b(v) >= f(v).
Mit gutem b(v): zu genau, Berechnung zu teuer, zu ungenau: kombinatorische Explosion

- **Best-first (Best First) Search:** Gehe dem Pfad mit dem höchsten Bound-Wert zuerst entlang.
- Korrigiere den b(v)-Wert des betrachteten Knotens anhand der Bound-Werte der darunter liegenden Knoten bzw. des erreichten Ergebnisses. Der Bound-Wert ist das Maximum der Bound Werte der direkten Nachfolger.
- Ist der gefundene Wert höher als die anderen Bound-Werte, dann haben wir die Lösung gefunden.
- Kombination aus Breitensuche und Tiefensuche.

Verbesserter Algorithmus: (Pruning, Cut off) Falls eine bereits gefundene Lösung besser ist, als die mittels der oberen Schranke b(v) geschätzte bestmögliche Lösung, dann muss dieser Teilbaum nicht mehr betrachtet werden.
Das «Abschneiden» eines Astes im Entscheidungsbaum wird als Pruning bezeichnet.

Branch and Bound (BB, BnB) Branch: Aufteilen der Lösung in mehrere, einfachere Teilschritte (z.B. mit einem Baum).
Bound: Durch das Festlegen von Schranken Teillösungen als nicht optimal kennzeichnen und nicht mehr verfolgen (abschneiden = Pruning).

Minmax - Algorithmus Entscheidungsbaum: alle möglichen Züge (untersucht kompletten Baum). Berechne für jede Endposition einen b(v)-Wert (auch Bewertungsfunktion bezeichnet). Zug wählen der den höchsten b(v) verspricht. Dabei entsteht **Horizont Effekt** Kann nur bis n ausgerechnet werden, ausgewählte Lösung könnte sich gleich danach als schlecht erweisen => nur von ausgewählter Lösung wird noch ein bischen mehr berechnet

Minmax Algorithmus mit Alpha-Beta Pruning Das Alpha-Beta-Pruning ist eine optimierte Variante des MinimaxSuchverfahrens: Findet bestimmt die beste Lösung

- Der Minimax-Algorithmus analysiert den vollständigen Suchbaum.
- Das Alpha-Beta-Pruning ignoriert alle Knoten, bei denen bereits bei der Suche feststeht, das dieser Ast das Ergebnis nicht beeinflussen kann

8.1.4 Komplexitätsklasse

Mittels Komplexitätsklassen (z.B. P oder NP) können Probleme (nicht nur Algorithmen) z.B. bezogen auf Rechenzeit klassifiziert werden. Dabei wird stets das kostengünstigste Lösungsverfahren (Algorithmus) angenommen.

9 Vorlesung 09

9.1 Suchen

Bsp: exits, count, find, compare...

9.1.1 Begriffe

- Vorbedingung: Aussage, die vor dem Ausführen der Programmsequenz gilt

- Nachbedingung: Aussage, die nach dem Ausführen der Programmsequenz gilt.
- Invariante: Aussage, die über die Ausführung bestimmter Programmbefehle hinweg hinweg gültig bleibt.
  - Invarianten können zum Beweis der Korrektheit von Alg. verwendet werden.
  - Verwendung im Design By Contract: Für Methoden/Klassen werden alle Vor- und Nachbedingungen und Invarianten in ihrem Ablauf beschrieben.

9.1.2 Binäres Suchen im sortierten Array

Gegeben sei ein sortiertes Array von Werten (Buchstaben). Wie kann ein Wert S in so einem Array effizient gesucht werden?

1. Führe zwei Indizes ein: l und r
2. Invariante: $\forall k, n; k \leq l, n \geq r; a[k] < S \wedge a[n] > S$
3. Wir verkleinern den Bereich zwischen r und l möglichst rasch, unter Einhaltung der Invariante. Spätestens wenn nur noch ein Element dazwischen liegt, haben wir S gefunden/nicht gefunden.

O(log(n)), bei jdm durchgang wird r-l halbiert.
Das Finden ist mit wenigen Schritten getan. Das Array muss aber sortiert sein (kostet etwas), sonst funktioniert das nicht!

9.1.3 Suchen: In mehreren Listen: Einfacher Algorithmus

Iteration durch 2 Arrays mit equals: O(n*m)

9.1.4 Suchen: In mehreren Listen: Besserer Algorithmus wenn a und b sortiert

Invariante: $\forall k, n; k < j, n < i; b[k] \neq a[n]$ Bereich der für Invariante gilt, wird sukzessive erweitert: O(n)

9.2 Hashing

Verfahren das (im idealfall) unabhängig von der Anzahl Elemente ist
Datenmenge wird so geformt: Schlüssel : Inhalt (= Schlüsseltabelle (Hashtable, selbst schwierig zu Hashen))
Hashing Idee: Werte in Array an ihrer Indexposition (z.B. bestimmt durch ASCII-Code) speichern. Suchen/Einfügen: O(1)

9.2.1 Usage

Damit Speicher gut genutzt wird kann nicht einfach ints oder Buchstaben direkt verwendet werden, es wird eine **Hashfunktion** verwendet, die X mod tableSize den Wertebreich auf kleineren Bereich abbildet.
Gute Hashfunktionen:

- $h(k) = kmodM$ [M ist häufig Primzahl]
- $h(k) = \lfloor m((kc)mod1) \rfloor$ [m = Anz. Hashadressen, 0 < c < 1; z.B. c = 1/gld. Schnitt = 0,618033...]

In Java gilt: Objekt muss für Lebensdauer immer denselben Hashwert haben, equals == true : Objekte haben denselben Hashwert, compareTo() == 0

9.2.2 Probleme

- Aus dem Hash-Wert kann der ursprüngliche Wert nicht mehr bestimmt werden. => Originalwert muss auch gespeichert werden
- Suche einer geeigneten Hash-Funktion. => Schwierig, Kenntnisse der zu erwartenden Schlüssel sinnvoll
- Zwei unterschiedliche Objekte können den gleichen Hash-Wert haben, d.h. sie müssten an der gleichen Stelle (Massierungen / Clustering) gespeichert werden => Kollision. => Kollisionen werden vermieden (Bildbereich gleich gross wie Ur-Bildbereich) - oder verringert (suche einer geeigneten Hash-Funktion). ODER Kollision wird aufgelöst, verschiedene Verfahren zur Auflösung (später): linear / quadratic probing, ...

- Wenn Hashtabelle voll, muss Hashtabelle vegrössert und alle Werte rehashed werden

9.2.3 Vor / Nachteile von Hashing

- Suchen Einfügen in Hash-Tabellen sehr effizient.
- ab 0.8 Belegungsgrad d. Schlüsseltabelle treten vermehrt Kollisionen auf
- «Einfache» binäre Bäume können degenerieren, Hash-Tabellen kaum.
- Implementationsaufwand für Hash-Tabellen geringer als für ausgeglichene, binäre Bäume.
- Kleinste oder größte Elemente schwer zu finden.
- Geordnete Ausgabe nicht möglich.
- Suche nach Werten in einem Bereich oder Finden z.B. eines Strings mit unbekanntem Anfang nicht möglich.

9.2.4 Kollisionsauflösung

Überlauflisten (Separate Chaining) Hashtable lediglich als Ankerpunkt für Listen aller Objekte, die den gleichen HashWert haben

Open Addressing Techniken, bei welchen bei einer Kollision eine freie Zelle sonst wo in der HashTable gesucht wird => setzt einen Load-Faktor < 0.8 voraus (sonst schlechte Performance)

- **Lineares Sondieren** (Linear Probing): Sequentiell nach nächster freier Zelle F+1, F+2, F+3, ..., F+i suchen (mit Wrap around). : i.d.R: O(1)
  - Phänomen des Primary-Clusterings (wenn viele Zeilen nebeneinander belegt werden müssen wirds nachher schlimmer)
  - Hash-Funktion: Input modulo Tabellengrösse
  - Bei zunehmendem LoadFaktor dauert es immer länger bis eine Zelle gefunden wird (Einfügen und Suchen)
  - Braucht aber nicht mehr Speicher als quadratisch
  - find funktioniert wie insert: Element wird in Tabelle ausgehend vom Hash-Wert gesucht bis Wert oder leere Zelle gefunden wird
  - bei hohem Load-Faktor oder ungünstigen Daten bricht die Performance ein
- **Quadratisches Sondieren** (Quadratic Probing): In wachsenden Schritten der Reihe nach F+1, F+4, F+9, . . . , F+i2 prüfen (mit Wrap around).
  - Hash-Funktion: Input modulo Tabellengrösse.
  - Jetzt bleiben Lücken in der Hash-Tabelle offen.
  - Bessere Performance als lineares Probing weil weniger Primary-Clustering auftritt.
- **Load Faktor** Sagt wie stark der Hash-Bereich belegt ist, Bewegt sich zwischen 0 und 1, Die Anzahl Kollisionen ist abhängig vom Load-Faktor λ und der Hash-Funktion h: f(h, λ)

Kollisionsauflösung Löschen Werte können nicht einfach gelöscht werden, da sie die Folge der Ausweichzellen unterbrechen:

1. Erste Möglichkeit: Wenn ein Wert gelöscht wird, müssen alle Werte, die potentielle Ausweichzellen sind, gelöscht und wieder eingefügt werden (rehashing).
2. Zweite Möglichkeit: Gelöschte Zelle lediglich als «gelöscht» markieren.

9.3 Extendible Hashing

Was tun, wenn die Hashtabelle überläuft oder sogar nicht mehr in den Hauptspeicher passt?

- Hashtabelle passt noch in den Hauptspeicher:
  - Überlaufketten → Performance beim Zugriff verschlechtert sich.
  - Umsiedlung in neue, ausreichend große Hashtabelle (Rehashing mit neuer Hashfunktion) → relativ teure Operation.

- Hashtabelle passt nicht mehr in den Hauptspeicher:
  - Extendible Hashings:
    - Schlüsselwertebereich kann nachträglich vergrößert werden.
    - Funktioniert gut mit Dateien/Blockstruktur (wie B-Bäume).

9.3.1 Grundidee

Verwende Hash-Verzeichnis von Verweisen zu «Buckets» (= Behälter)

- Ein Bucket enthält mehrere Einträge.
- Die Buckets müssen nicht mehr hintereinander auf der Disk liegen.
- Grosse Buckets (z.B. Bucket-Size = Disk-Block) haben kleine Schlüssel im Hash-Verzeichnis zur Folge.
- Hash-Verzeichnis enthält lediglich (die letzten) n-Bits des Schlüssels und Hash-Verzeichnis ist damit wesentlich kleiner (als die Daten selbst)
- Das Verzeichnis kann in den Hauptspeicher geladen werden.
- Es kann einfach verdoppelt werden (ein Bit mehr berücksichtigen).
- Insert: Falls ein Bucket voll ist (Overflow), dann Bucket teilen. Falls notwendig (globale Tiefe = lokale Tiefe), dann auch das Verzeichnis verdoppeln.

9.4 Hashing in Java

Map und HashMap

10 Vorlesung 10

10.1 Suchen in Texten: Java

Text: String, StringBuilder, StringBuffer: indexOf()

10.2 Brute Force

Idee: Pattern wird startend von Position 0 nach und nach verschoben und jeweils mit String verglichen

- Brute Force: Worst Case O(Anz Zeichen String*Anz Zeichen Pattern)
- Brute Force Verbessert: Es wird jew. der erste Buchstabe gesucht bevor ganzes Pattern gesucht wird
- Brute Force Verbessert: Es wird jeweils um die Länge des Patterns vorgerückt anstatt nur um 1 (Dabei muss aber SubPattern (sich wiederholende innerhalb des Patterns beachtet werden: Knuth Morris Pratt Algorithm:
  1. Zuerst wird Pattern durchsucht nach Wiederholungen und eine Next-Tabelle gebildet,
  2. die dann jeweils sagt um wieviel verschoben wird bei non match
  3. O(n+m) Laufzeit

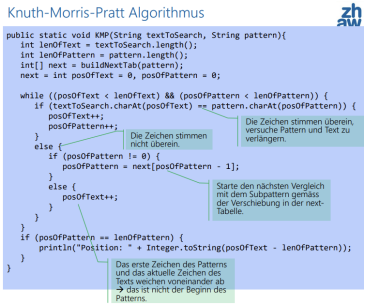

Abbildung 2: Knuth-Morris-Pratt Algorithmus

10.3 Invertierender Index

ein invertierter Index ist eine Indexdatenstruktur, die eine Abbildung vom Inhalt, wie z.B. Wörter oder Zahlen, auf seine Positionen ...in einem Dokument oder einer Gruppe von Dokumenten speichert

- Wird häufig v. Suchmaschinen verwendet

- Der invertierte Index kann z.B. als B-Baum implementiert werden.
- mit dem Invertierten Index lässt sich Häufigkeit von einzelnen Wörtern gut bestimmen
- Web Roboter/Spider/Crawler durchsuchen das Web nach neuen Informationen
- **Indexierung** Aufbreitung von Dokumenten, Speicherung in Index/der Datenbank der Suchmaschine
- Retrievalsystem: Suchen im Index, Sortierung nach Relevanz
- O(n) falls Wörter unsortiert, kann mit sortieren und binär, balanciertem Baum, Hashing, nicht relevante Wörter entfernen, Wortstämme bilden verbessert werden

10.4 Levenshtein Distanz (Approximative Suche)

Die Levenshtein-Distanz (auch: Editier-Distanz) von zwei Wörtern A und B ist die minimale Anzahl Operationen, um aus dem ersten Wort das zweite Wort zu machen: Suche von ähnlichen Wörtern, Wörter mit kleinen Schreibfehlern

- Der Aufwand zur Berechnung der Levenshtein-Distanz ist O(n+m); n = Länge des ersten Wortes, m = Länge des zweiten Wortes.
- Die Levenshtein-Distanz gibt an, wie teuer"die Umwandlung eines Wortes in ein anderes Wort ist.

Erlaubte Operationen:

- insert(c): Buchstaben ’c’ an einer Position im ersten Wort einfügen.
- update(c→d): Buchstaben ’c’ an einer Position im ersten Wort durch ’d’ ersetzen.
- delete(c): Buchstabe ’c’ an einer Position im ersten Wort löschen.

Algorithmus für kürzeste Distanz Gegeben zwei Wörter A und B der Länge n und m.
Konstruiere eine Matrix D mit der Grösse (n+1)·(m+1).
D[i, j] gibt die Levenshtein-Distanz der Präfixe von A und B der Länge i und j an.

10.5 Trigramm-Suche

- Fehlertolerante Suche (auch für Wortverdreher, z.B. Vor- und Nachname).
- Effizient für grosse Datenbestände.
- Index => Wort in 3er-Buchstaben Gruppen unterteilt:
  - Z.B. sind das bei «Peter», drei 3-er Gruppen «PET», «ETE», «TER».
  - Diese 3-er Gruppen werden für vorkommende Worte gebildet und z.B. in der Hashtabelle gespeichert.
- Das zu suchende Wort wird ebenfalls in 3-er-Gruppen zerlegt.
- Das gesuchte Wort mit am meisten Übereinstimmungen wird genommen.

10.6 Phonetische Suche: Soundex

Ein Wort besteht z.B. aus seinem ersten Buchstaben, gefolgt von drei Ziffern, z.B. "K523"(Soundex-Code):

- Die Vokale A, E, I, O und U und die Konsonanten H, W und Y sind ausser beim ersten Zeichen zu ignorieren (in D auch ä, ö, ü).
- Kurze Worte: mit 0 auffüllen, 0-werden ignoriert
- Ziffern sind Konsonanten nach folgender Tabelle

Englisch

Ziffer	Repräsentierte Buchstaben
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Deutsch

Ziffer	Repräsentierte Buchstaben
0	a, e, i, o, u, ä, ö, ü, y, j, H
1	b, p, f, v, w
2	c, g, k, q, x, s, z, ß
3	d, t
4	l
5	m, n
6	r
7	ch

Abbildung 3: Soundex Tabelle

10.7 Suche Nach Mustern: Regex

Zum Suchen von definierten Mustern in Texten (Bestimmter String, Unscharfe Muster, Wiederholende (Teil-)Muster
Klassen: Pattern und Matcher

11 Vorlesung 11

11.1 Motivation

Sortieren als eigenständige Aufgabe (Worte, Dateien, Programm, Karten, etc) oder zur Steigerung der Effizienz eines Algorithmus (Binäre Suche, SQL Befehle)

11.2 Gleiche Objekte finden

11.2.1 Algorithmus 1

Jedes Objekt mit jedem Vergleichen: $O(n^2)$

11.2.2 Algorithmus 2

Objekte sortieren O(n) und dann benachbarte Vergleichen O(n*log(n)) = O(n*log(n))

11.3 Internes vs Externes Sortieren

11.3.1 Internes Sortieren

Wenn die Anzahl der Datensätze und deren jeweiliger Umfang sich in Grenzen halten, kann man alle Datensätze im Arbeitsspeicher eines Computers sortieren.
Man spricht dann von einem internen Sortiervorgang (Engl. internal sort):
Vor dem Sortieren werden alle Daten in RAM geladen, dann sortiert, dann wieder gespeichert

11.3.2 Externes Sortieren

Können nicht alle Datensätze gleichzeitig im Arbeitsspeicher gehalten werden, dann muss ein anderer Sortieralgorithmus gefunden werden.
Man spricht dann von einem externen Sortiervorgang (Engl. external sort).
Teilen der grossen zu sortierende Datei in n Teile (klein genug, dass sie in den Hauptspeicher passen).
Dateien werden nacheinander in Speicher (parallel?) eingelesen, intern sortiert und wieder in Dateien geschrieben.
Die sortierten Dateien werden schliesslich zu einer sortierten Datei zusammengefügt (merge).
Das Problem des externen Sortierens lässt sich auf das des internen Sortierens zurückführen bzw. setzt voraus, dass ein Teil der Daten intern sortiert werden kann.

11.4 Sortierschlüssel: Sortieren von Datensätzen

Aufbau: Sortierschlüssel + Inhalt
Der Sortierschlüssel ist ein Teil des Inhaltes.
Der Sortierschlüssel kann aus einem oder mehreren Teilfeldern bestehen, für die eine sinnvolle Ordnung gegeben ist.
Bei Textfeldern kann dies eine lexikographische Anordnung sein - bei Zahlen eine Anordnung entsprechend ihrer Grösse.
Der übrige Inhalt der Datensätze ist beliebig und wird nicht weiter betrachtet.

11.4.1 Sortierschlüssel Definition

Def: Sortierschlüssel sind Kriterien, nach denen Datensätze sortiert oder gesucht werden können
Kann eindeutig sein, kann Kombination aus verschiedenen Feldern sein (Bsp. name, vorname, alter)

11.4.2 Sortierschlüssel Eigenschaften

Für Sortierung ist kein eindeutiger Sortierschlüssel notwendig, doch ist er nur sinnvoll, wenn er den Datensatz weitgehend bestimmt.

Beim Anlegen einer relationalen Datenbank sollte dagegen in jeder Tabelle ein eindeutiger Schlüssel vorhanden sein

11.4.3 Sortierschlüssel Vergleich

Um Datensätze sortieren zu können, müssen wir die Schlüsselwerte entsprechend der gewählten Ordnungsrelation vergleichen können. (Strings /Zahlen können direkt verglichen werden, für Kombi Schlüssel muss Comparable / Comparator implementiert werden)

11.4.4 Collation : Alphabetische Sortierung

Computer Systeme ordnen jedem Buchstaben einen Code zu, z.B. ASCII, Unicode, EBCDIC. Resultat des (String-)Vergleichs ist durch diese Ordnung festgelegt: A..Z,..,a..z
In Auswertungen (z.B. Telefonbüchern) wird aber oft eine länderspezifische Sortierung gefordert.
Mittels einer Collation (dt. Kollation = Einsortierungsregeln) kann eine spezifische Sortierreihenfolge festgelegt werden.

- Collator-Klasse (java): implementiert Comparator-Interface und führt mit getInstance() Ländereinstellung des Systems durch
- Locale-Konstruktioren: locale(String Language, Contry, Variant)

11.5 Die drei einfachen internen Sortierverfahren

Es wird nur nach dem Schlüssel sortiert, dh. der Inhalt der sortierten Daten spielt keine Rolle, die Art des Schlüssels auch nicht.

Sortier-Algorithmen Hilfsmethode: In den meisten Algorithmen werden Elemente vertauscht, es wird deshalb im Folgenden die Existenz folgender swap-Methode angenommen:
private static <K> void swap(K[] kArray, int i, int j) {
K h = kArray[i]; kArray[i] = kArray[j]; kArray[j] = h;
}

11.5.1 Insertion Sort (stabil)

Der Spieler nimmt eine Karte nach der anderen auf und sortiert sie in die bereits aufgenommenen Karten ein.
Idee: Teile den Bereich in 2 Teile auf (sortiert / unsortiert):
Invariante (Start k = 0): $\forall n; n > 0 \wedge n < k; a[n-1] \leq a[n]$

- Wir entnehmen das Element ganz links vom unsortierten Teil.
- Die entstehende Lücke wird nach links verschoben, bis die korrekte Position für das Element gefunden wurde.
- Dort wird das Element eingeordnet.
- Resultat pro Durchlauf: Der sortierte Bereich wurde um ein Element vergrössert.
- Aufwand: mit k = Aufwand für Vergleich und swap Anweisungen in innerer Schleife $k((n-1)+(n-2)+...+2+1) = kn(n-1)1/2$
  - Best Case: $k*(n-1) = \Omega(n)$
  - Average Case: $(3/8)*k*n*(n-2) = \Theta(n^2)$
  - Worst Case: $kn(n-1)1/2 = O(n^2)$

11.5.2 Selection Sort (instabil)

Der Spieler nimmt die jeweils niedrigste der auf dem Tisch verbliebenen Karten auf und kann sie in der Hand links (oder rechts) an die bereits aufgenommenen Karten anfügen.
Idee: Teile den Bereich in 2 Teile auf (sortiert / unsortiert):
Invariante (Start k = 0): $\forall n; n > 0 \wedge n < k; a[n-1] \leq a[n]$

- Suche jeweils das kleinste der verbleibenden Elemente und ordne es am Ende der bereits sortierten Elemente ein.
- In einem Array a mit dem Indexbereich 0..h sei k die Position des ersten Elements im noch nicht sortierten Bereich und i die Position des kleinsten Elementes in diesem Bereich.

- Wenn wir nun a[k] und a[i] vertauschen, dann haben wir den sortierten Bereich um ein Element vergrössert.
- Wenn wir diesen Vorgang so lange wiederholen, bis k = a.length gilt, ist das ganze Array sortiert.
- Aufwand: [mit k1 = Aufwand Vergleich, k2 = Aufwand swap]: $k1n(n-1)1/2 + k2(n-1)$
  - Aufwand für alle Fälle gleich: $O(n^2)$
- Deutlich wengier Swap-Aufrufe als Bubblesort, kann dafür Vorsortiertheit nicht ausnutzen

11.5.3 Bubble Sort (stabil)

Der Spieler nimmt alle Karten auf, macht einen Fächer daraus und fängt jetzt an, die Hand zu sortieren, indem er benachbarte Karten so lange vertauscht, bis alle in der richtigen Reihenfolge liegen.

- Nach dem 1. Durchgang hat man die folgende Situation:
  - Das grösste Element ist ganz rechts.
  - Alle anderen Elemente sind zwar zum Teil an besseren Positionen (also näher an der endgültigen Position), im Allgemeinen aber noch unsortiert.
- Der Array wird in mehreren Durchgängen von links nach rechts durchwandert.
- Dabei werden Nachbarfeldern, die in falscher Reihenfolge stehen, vertauscht.
- Dies wiederholt man so lange, bis der Array vollständig sortiert ist.
- Das Wandern des grössten Elementes ganz nach rechts kann man mit dem Aufsteigen von Luftblasen in einem Aquarium vergleichen (BubbleUp).
- if (a[i] > a[i + 1]) swap (a, i, i + 1);
- Meist sind Daten bereits vor dem letzten Durchgang sortiert, dann kann abgebrochen werden
  - Bei jedem Durchgang testen, ob überhaupt etwas vertauscht wurde.
  - Wenn in einem Durchgang nichts mehr vertauscht wurde, sind wir fertig.
- Aufwand: mit k = Aufwand für Vergleich und swap Anweisungen in innerer Schleife $k((n-1)+(n-2)+...+2+1) = kn(n-1)1/2$
  - Best Case: $k*(n-1) = \Omega(n)$
  - Ist d. schnellste Alg. wenn Daten bereits sortiert.
  - Average Case: $(3/8)*k*n*(n-2) = \Theta(n^2)$
  - Worst Case: $kn(n-1)1/2 = O(n^2)$

	Best-Case:	Average-Case:	Worst-Case:
Bubble-Sort:	k · ((n-1)	(3/8) · k · n · (n-2)	k · n · (n-1) · ½
Selection-Sort:	$k_1 \cdot n \cdot (n-1) \cdot ½ + k_2 \cdot (n-1)$		
Insertion-Sort:	k · ((n-1)	k · n · (n-1) · ¼	k · n · (n-1) · ½
Bubble-Sort:	Ω(n)	Θ(n^2)	O(n^2)
Selection-Sort:	Ω(n^2)	Θ(n^2)	O(n^2)
Insertion-Sort:	Ω(n)	Θ(n^2)	O(n^2)

Abbildung 4: Vergleich Sortierungsalgorithmen Laufzeit und Ordnung

11.6 Sortierung: Ordnung vs. Laufzeit

Die **Ordnung** besagt, wie stark sich der Aufwand bei einer Veränderung der (Anzahl der) Eingangsdaten verändern. O(n2): Verdoppelung von n => Aufwand wird 4-mal grösser
Die **Laufzeit** besagt, wie lange das Programm benötigt. Ist von vielen Faktoren abhängig, wie Rechnergeschwindigkeit, verwendeter Programmiersprache, Cache, Compilereinstellungen, bei MehrprozessBetriebssystemen: auch Auslastung der Maschine und Priorität des Prozesses.

Laufzeit kann für unbekannte n angenähert werden: $O(n^2)$ => Polynom 2. Grades: $k1 * n^2 + k2 * n + k3$, Messen von verschiedenen Laufzeiten, nach den k's auflösen, bei grossen n wird oft nur k1 bestimmt.

11.7 Sortieralgorithmen: Stabilität

Einmal sortierte Folgen behalten ihre Reihenfolge.
Ein wichtiger Punkt bei Sortieralgorithmen ist die Art wie Elemente mit gleichem Schlüssel behandelt werden.
Sei S = ((k0, e0), ... , (kn-1, en-1)) eine Sequenz von Elementen:
Ein Sortieralgorithmus heisst **stabil (stable)**, wenn für zwei beliebige Elemente (ki, ei) und (kj, ej) mit gleichem Schlüssel ki = kj und i < j (d.h. Element i kommt vor Element j), i < j auch noch nach dem Sortieren gilt (Element i kommt immer noch vor Element j)

12 Vorlesung 12

12.1 Teile und Herrsche (TUH, divide et impera)

Zerlege das Problem in kleinere, einfache zu lösende Teile.
Spezialfall: Teil = Ursprungsproblem mit kleinerem Bereich.
Löse die so erhaltenen Teilprobleme. Und führe danach wieder zum ganzen zusammen

```
if (Menge der Datenobjekte klein genug)
   Ordne sie direkt;
else {
   Teilen: Zerlege die Menge in Teilmengen;
   Ausführen: Sortiere jede der Teilmengen;
   Vereinigen: Füge die Teilmengen geordnet zusammen;
}
```

TUH-Algorithmen sind typischerweise rekursiv:

```
Sort (Menge a)
   if (Menge der Datenobjekte klein genug)
      Ordne sie direkt;
   else {
      Zerlege in zwei Teilmengen;
      Sort(Teilmenge1); Sort(Teilmenge2);
      Füge Teilmengen geordnet zusammen;
   }
}
```

Bei der Zerlegung sollten die Teile möglichst gleich gross sein.

Abbildung 5: Teile und Herrsche Sortieralgorithmen

12.2 Quick-Sort

mit naheliegenden Verbesserungen ist einer der schnellsten bekannten allg. Sortieralgorithmen
Die Grundidee besteht darin, das vorgegebene Problem nach dem bereits genannten Motto Teile und Herrsche in einfachere Teilaufgaben zu zerlegen:

- Nehme irgendeinen Wert W der Teil des Sortierfeldes A ist – zum Beispiel den Mittleren.
- Konstruiere eine Partitionierung von A in Teilmengen A1 und A2 mit folgenden Eigenschaften:
  - $A = A1 \cup A2 \cup W$
  - Alle Elemente von A1 sind $\leq$ W (aber evtl. noch unsortiert).
  - Alle Elemente von A2 sind $\geq$ W (aber evtl. noch unsortiert).
- Wenn jetzt A1 und A2 sortiert werden, ist das Problem gelöst.

12.2.1 Partionierung (Wahl des Pivots)

Noch zu lösende Probleme der Partitionierung:
1. Wahl eines Pivotwertes W (Franz. Pivot = Drehpunkt), so dass A1 und A2 möglichst gleich gross sind.
2. Verteilen von A auf A1 und A2

- Pivotwahl ist von entscheidender Bedeutung für die Effizienz von Quick-Sort.
- Optimal wäre ein Element, das A in zwei gleich grosse Teile partitioniert.
- W sollte so bestimmt werden, dass gleich viele Werte grösser und kleiner als W sind (Median)
- Gutes Pivot-Element ist nicht unbedingt das Element, welches der Position nach aktuell in der Mitte liegt.
- Möglichkeit einer ungünstigen Verteilung der Daten: Durch die Partitionierung können in eine Hälfte der Partition sehr viele und in die andere Hälfte sehr wenige Daten gelangen.

- Bestimmen des genauen Median-Elementes aufwendig => Laufzeitvorteil von Quick-Sort ginge wieder verloren.
- W wird lediglich «geschätzt».
  - A[l] das (der Position nach) linke Element von A;
  - A[r] das (der Position nach) rechte Element von A;
  - A[mid] das (der Position nach) mittlere Element von A mit mid = (l+r)/2
  - Strategie 1: Nehme eines der drei Elemente.
  - Strategie 2: Nehme das (wertmässig) mittlere der drei Elemente (Median).

12.2.2 Partionierung (Verteilung von A)

- Gewähltes Pivotelement an den Rand verschieben:
  - Wähle einen Wert W (Pivotelement) aus A.
  - Schiebe das Element W ganz nach rechts.
- Elemente austauschen ($\neq$ sortieren), A1 und A2 erstellen:
  - Überprüfe von ganz links kommend (j), ob das Element in den linken Bereich (A1) gehört, d.h. A[j] < W.
  - Falls ja, vertausche das Element A[j] mit dem Element A[i] (i ist der nächste freie Platz in der Partition A1) und erhöhe i um 1.
  - Wiederhole obige Schritte so lange j $\leq$ n-1
- 3. Pivotelement an korrekten Platz (i) verschieben
- Rekursiv wieder oben anfangen für A1 und A2

12.2.3 Aufwand

Die Rekursionstiefe ist log2(n), wenn bei jeder Partitionierung eine gleichmässige Aufteilung der Daten erfolgt:

- Es entsteht dabei ein binärer Partitionenbaum mit Tiefe log2(n).
- Der Aufwand auf jeder Schicht, diese komplett zu partitionieren, ist proportional zu n.
- Der Gesamtaufwand ist somit proportional zu n·log2(n).
- Die Ordnung von Quick-Sort ist in diesem Fall daher O(n·log(n)).
- Ist ein Inplace-Verfahren (braucht keinen zusätzlichen Speicherplatz)

Ein Sortieralgorithmus, der darauf beruht, dass Elemente untereinander verglichen werden, kann im Worst-Case bestenfalls eine Komplexität von O(n·log(n)) haben.
Aufwand im ungünstigsten Fall:

- Jeder Partitionierungs-Schritt enthält eine leere Partition => Baum wird zu Kette (Liste) mit n Elementen: man sagt auch der Baum entartet oder degeneriert.
- In diesem Fall ist der Aufwand proportional zu O(n2).
- Dies tritt jedoch nur in extra konstruierten Fällen auf, oder wenn man Pech hat. Im Normalfall ist die Partitionierung bei Quick-Sort nahezu optimal.

12.2.4 Optimierung von Quick-Sort

Für wenige ($\sim$ 10..1000) zu sortierende Daten ist ein einfaches Sortierverfahren schneller.
Idee: Unterhalb einer bestimmten Länge des Intervalls nicht mehr Quick-Sort verwenden, sondern z.B. Insertion-Sort; bringt ca. 10%

12.3 Distribution-Sort

- Die bisher diskutierten Sortieralgorithmen basieren auf den Operationen: Vergleichen zweier Elemente und ev. Vertauschen zweier Elemente (swap).
- Im Gegensatz dazu kommt Distribution-Sort ohne Vergleiche zwischen den Elementen aus.
- Zu sortierende Elemente werden (Teile und Herrsche):

  - entsprechend dem Sortierschlüssel in Fächer verteilt: O(n)
  - zusammengetragen: O(n)
  - => Gesamtaufwand O(n)
- Bsp.: Briefe werden in die entsprechenden Fächer nach Postleitzahl sortiert.

12.3.1 Aufwand

Grundprinzip wie beim direkten Adressieren (vergl. Hashtable):

- Vorteile:
  - Schneller geht's nicht.
  - Linearer Algorithmus: die Komplexität ist also O(n).
- Nachteile:
  - Verfahren muss an den jeweiligen Sortierschlüssel angepasst werden.
  - Geht nur bei Schlüsseln, die einen kleinen Wertebereich haben, oder auf einen solchen abgebildet werden können, ohne dass die Ordnung verloren geht.
  - Allgemeines Hashing funktioniert nicht
- Distribution-Sort ist mit Abstand der schnellste Algorithmus zum Sortieren.
- Es handelt sich aber nicht um ein allgemein anwendbares Sortierverfahren.

12.4 Merge-Sort: Extern

Beim externen Sortieren liegen die Daten in einer Datei (z.B. auf der Festplatte). Zwei Arten des Zugriffs sind möglich:

- Sequentieller Zugriff.
- Der beliebige Zugriff auf die Elemente wäre zwar möglich, ergibt aber einen grossen Effizienzverlust.

Annahmen:

- Datenstrom wird sequentiell gelesen.
- Jeweils nur ein Teil der Daten passt in den Hauptspeicher.

12.4.1 Merge Sort Anleitung

- Lade jeweils einen Teil der Datei in den Speicher (Teile und Herrsche).
- Sortiere diesen Teil mit schnellem internem Verfahren.
- Schreibe diesen Teil sortiert in eine die Ausgabedateien. Es entstehen Folgen von sortierten Abschnitten in den Ausgabedateien.
- Lese von beiden Dateien das jeweils erste Element.
- Schreibe das Kleinere in den Output und lese das Nächste von der gleichen Datei.
- => Länge der geordneten Abschnitte hat sich verdoppelt. So lange wiederholen, bis vollständig zusammengefügt.

12.4.2 Aufwand

Annahmen:

- Zeit für internes Sortieren kann vernachlässigt werden.
- Beispiel: 16 Sequenzen (sortierte Abschnitte aus Sort-Phase), je 2 Eingabe-Dateien während Mischphase => log2(16) Mischphasen.

Bei insgesamt n Elementen und m Eingabedateien:

- Aufwand Sortierphase (Erstellen der Dateiteile): n
- Aufwand je Mischphase: n
- Anzahl Mischphasen: log2(m)
- Gesamtaufwand Sortieren und Mischen: n + n·log2 (m) => O(n·log(m))

Der Merge-Sort kann auch als internes Sortierverfahren (z.B. auf Arrays) angewendet werden.

- Der Aufwand beträgt in diesem Fall O(n·log(n)), da n=m.
- Das Verfahren benötigt jedoch zusätzlichen Speicherplatz der Grössenordnung O(n), ist also kein Inplace-Verfahren.
- Das Verfahren ist im Gegensatz zum Quick-Sort stabil

12.5 Wahl des Sortierverfahrens

Auswahlkriterien des geeigneten Sortierverfahrens:

- Internes oder externes Verfahren
  - Sortieren im Hauptspeicher
  - Sortieren im Hauptspeicher und Sekundärspeicher (Platte/SSD)
- Methode des Algorithmus:
  - Vertauschen / Auswählen / Einfügen
  - Rekursion
  - Mehrphasen: Sortieren-Mischen
- Nach Effizienz:
  - Laufzeit: O(n2) oder O(n·log(n)) oder sogar O(n)
  - Speicherbedarf/Inplace: Wieviel Speicher wird zusätzlich zu dem für die zu sortierenden Daten benötigt?

Auswahl

- Stabilität: Wird die Reihenfolge von Datensätzen mit gleichem Sortierkriterium durch den Algorithmus geändert?
- Der Algorithmus setzt eine bestimmte Struktur der Schlüssel voraus (z.B. Distribution-Sort).
- Wenige Datensätze (weniger als 1'000), für 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 angepasst.
- Sehr viele Daten: externes Sortierverfahren in Kombination mit schnellem internem Sortierverfahren.

12.6 Parallelisierung: Speed vs Power Consumption

- Das grösste Problem heute ist die Wärmeabgabe der Chips.
- Jeder CMOS Schaltvorgang braucht Energie, Verkleinerungen erhöht die Leckströme.
- Die Energiedichte innerhalb einer CPU ist grösser als diejenige im Kern eines Kernreaktors.

Amdahls Law: Parallelisierung: Beschleunigung

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}}$$

Mit [p = ist der Anteil des Programms der parallelisiert werden kann, s: Zahl der Prozessoren]

12.7 Java-Memory-Modell (JMM)

JMM abstrahiert von physikalischem Memory-Modell. Das JMM gibt an:

- Wie und wann verschiedene Threads Werte sehen können, die von anderen Threads in gemeinsame Variablen geschrieben wurden.
- Wie der Zugriff auf gemeinsame Variablen bei Bedarf synchronisiert werden kann.

Im JMM hat jeder Thread:

- Eigenen Thread-Stack, der Zugriff auf andere Thread-Stacks ist nicht möglich.
- Eigene Informationen über ausgeführte Methoden (Call Stack).
- Eigene lokale Variablen für primitive Datentypen jeder Methode (Stack).
- Objekte liegen im gemeinsamen Heap – nicht im Stack.

12.7.1 Naiver Quicksort

Je ein neuer Thread für jede Partitionsaufgabe.
Java-Threads werden auf Betriebssystem-Threads abgebildet: «Kernel Level Threads» (es gibt mehr Threads als Kerne)

- Vorteile:
  - Die Rechenzeitzuteilung kann besser (von BS) verwaltet werden.
  - Sind mehrere CPU-Kerne im Rechner vorhanden, können diese ausgenutzt werden.
- Aber:
  - Erzeugung und Zerstörung von Threads kostet (viel) Zeit.
  - Instanziierte Threads belegen Speicher.
  - Ineffektiv, mehr Threads zu erzeugen, als die Prozessoren gleichzeitig handhaben können, da immer einige inaktiv (idle) sein werden.
  - Quick-Sort eignet sich relativ schlecht für parallele Ausführung, da der parallelisierbare Anteil klein ist (Amdahl's Gesetze). Allenfalls ist Performance sogar schlechter...

13 Vorlesung 13

13.1 Thread-Pool: Threads and Tasks

Es wird einer gewissen Anzahl von einmalig gestarteten Threads (Pool von Threads) immer wieder eine neue Aufgabe (Task) übergeben.
Pool-Grösse entspricht in etwa der Anzahl Rechnerkerne: (Bei I/O-intensiven Tasks mehr, da immer ein Teil wartend/blockiert ist.)
Die Reihenfolge der Abarbeitung der Tasks wird durch eine Ausführungsstrategie bestimmt (Thread-Pool-Typ).

13.2 Thread-Pool: Ausführungsstrategien

Bessere Kontrolle über Ressourcen durch Einhaltung einer Ausführungsstrategie (Thread-Pool-Typ):

- In welcher Reihenfolge werden die Tasks in der Queue ausgeführt (LIFO, FIFO, priorisiert, ...).
- Anzahl der Tasks die gleichzeitig, parallel ausgeführt werden dürfen.
- Anzahl der Tasks die auf Ausführung warten dürfen.
- Periodische Threads (zeitgesteuert).

13.3 Thread-Pool: Java Implementation

Interface Executor (java.util.concurrent) stellt Framework zur Thread-Ausführung bereit, das verschiedene Ausführungsstrategien realisiert.
Interface ExecutorService für Thread-Pools.
Verschiedene Varianten des ThreadPoolExecutor werden durch Factory-Methoden in der Klasse (nicht Interface) Executors bereitgestellt (Fixed, Cached-ThreadPool / Single, Scheduled:ThreadExecutor)

13.3.1 Thread-Pool: Lebenszyklus

running: nimmt Tasks engegen und führt sie aus sobald Threads verfügbar sind
shutdown / stop: graceful / abrupt shutdown (Threadpool führt laufende und bereits angenommene, aber nicht begonnene Tasks noch aus, nimmt jedoch keine neuen Tasks mehr an. / Versucht, alle aktiven Task zu stoppen, stoppt die Verarbeitung von wartenden Tasks und gibt eine Liste der Tasks zurück, die auf ihre Ausführung warteten.
terminated: Keine Tasks werden mehr ausgeführt oder angenommen

13.3.2 Thread-Pool: Wie kommen Thread in den Pool

Einstellen eines Threads in Threadpool: Future<> submit
Verwendung von Callable-Interface wenn Rückgabewerte/Resultate benötigt werden (Runnable-Interface liefert NULL-Wert bei Erfolg).
Ein Future (Interface) stellt das Ergebnis einer asynchronen Berechnung dar

13.3.3 Thread-Pool: Resultat der Ausführung

Mit get()-Methode zur Abfrage des Rückgabewerts vom Typ V:

- Wartet, bis Rückgabewert feststeht.
- Ggf. nur so lange, bis angegebenes Timeout erreicht.

mit cancel()-Methode zur Stornierung der Aufgabe:

- unmittelbare Streichung, wenn Bearbeitung noch nicht begonnen.
- Abbruchversuch, wenn schon in Bearbeitung.

13.3.4 Thread-Pool: Thread-Safe-Collections

Im Paket java.util.concurrent gibt es einige Thread-Safe-Collections. Im Fall von erwarteten Race-Conditions (oder ähnlichem) sollten diese Klassen verwendet werden:

- ConcurrentHashMap<K,V>
- ConcurrentLinkedQueue<E>
- ConcurrentLinkedDequeue<E>
- SynchronousQueue<E>

13.4 Fork/Join

Eignet sich insbesondere für das «Teile und Herrsche»-Prinzip, rekursiv auf Parallelität angewandt.
ForkJoinPool macht grundsätzlich dasselbe wie ThreadPoolExecutor. Threads aber manchmal immer noch zu schwergewichtig

13.4.1 Fork/Join: Anleitung

1. Definiere ein ForkJoinPool-Objekt.
2. Definiere ein Task-Objekt, das von der Klasse RecursiveTask (falls mit Resultat) erbt.
3. Instanziere den ForkJoinPool.
4. Instanziiere (Root-)Task-Objekt und rufe invoke() auf.
5. In der compute()-Methode (innerhalb Task-Objekt) Code mit der Aufteilung des Problems:
   (a) Löse Problem direkt falls klein/einfach genug.
   (b) Sonst: Teile Problem auf: Rufe invokeAll(task1, task2, ...) auf; startet alle Tasks und wartet bis alle beendet sind. Oder: Starte Tasks mit fork() mit abschliessenden join() und mit invoke() beim Letzten

Threads	Fork/Join
subclass Thread	**subclass** RecursiveTask/Action<V>
override run	**override** compute
call start	**call** invoke, invokeAll, fork
call join (je Thread)	**call** join which returns answer or **call** invokeAll on multiple tasks

Abbildung 6: Threads vs ForkJoin

13.5 Umgang mit dem Hauptspeicher

13.5.1 Dynamische Speicherverwaltung (Heap)

Fehler beim Umgang mit dem dynamischen Speicher sind schwer zu entdecken, weil sie erst zur Laufzeit auftreten und ist für Effizienz des Programms oft ausschlaggebend
In objektorientierten Sprachen werden Objekte kreuz und quer referenziert, so dass es schwierig festzustellen ist, ob ein Objekt noch verwendet wird. Java kennt deshalb die automatische Speicherfreigabe: Garbage-Collection (GC, Müllsammler), bzw. Gargabe-Collector.

13.5.2 Java Memory Model

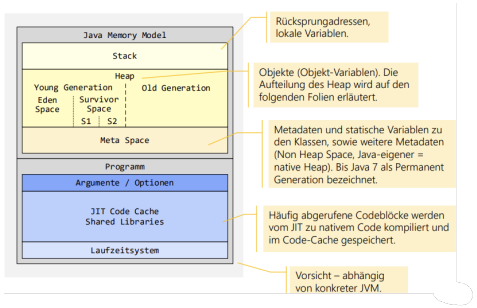

Abbildung 7: Java Memory Model

Java Memory Model Meta Space Einfachste Zuteilungsstrategie: Der Compiler legt die Zuordnung von Variablenname zu relativen Speicheradressen fest. Speicher wird beim Start des Programms angefordert und beim Beenden wieder freigegeben.
Meta Space enthält Klassendefinitionen (Class Object) inklusive deren statischer Variablen.

- einfach, bereits beim Start kann gesagt werden ob Speicher ausreicht
- grössen aller Datenstrukturen müssen zur übersetzungszeit bekannt sein, keine rekursiven Programmaufrufe / Dynamischen Datenstrukturen (Listen / Bäume) möglich

Java Memory Model Stack Bei jedem Aufruf einer Methode wird ein Speicherbereich reserviert: Frame.
Frames werden als Stack organisiert: LIFO.
Die lokalen Variablen der Methoden werden relativ zum Framepointer angesprochen.

- rekursive Aufrufe möglich, effizient in Zuteilung/Freigabe Speichers
- grösse der einzelnen Datenstrukturen muss bekannt, Werte auf dem Stack sind nach dem Verlassen der Methode verloren, Stack Overflow möglich

Java Memory Model Heap Der Heap verwaltet die Instanzen von Klassen (Objekte) und Arrays.
Der Speicher muss vom Programm mittels new angefordert und, je nach Sprache, mittels delete explizit wieder freigegeben werden.

- Die Grösse der einzelnen Datenstrukturen kann zur Laufzeit festgelegt werden, Dynamische Datenstrukturen, z.B. Listen, sind möglich.
- Zuteilung/Freigabe der Daten ist relativ rechenintensiv und kompliziert, Speicher muss explizit vom Programm angefordert und wieder freigegeben werden, Programmierfehler möglich (Memory Leak)., Speicherüberlauf (Heap-Overflow) möglich.

13.6 Einfache Speicherverwaltung: Freie Zuteilung

- Der Speicherverwalter unterhält zwei Listen:
  - Belegt-Liste: Liste der belegten Speicherbereiche.
  - Frei-Liste: Liste der freien Speicherbereiche.
- Speicher wird angefordert:
  - Ein Block der angeforderten Größe aus dem freien Bereich wird als belegt markiert.
  - Er wird in eine Belegt-Liste eingetragen.
  - Der Rest des Bereichs (Verschnitt) wird in die Frei-Liste eingetragen.
  - Eine Referenz auf den Bereich wird zurückgegeben.
- Speicher wird freigegeben:
  - Der Speicher wird aus der Belegt-Liste entfernt und in die Frei-Liste eingetragen.
- Problem:
  - Im Hauptspeicher bilden sich mit der Zeit Löcher (externe Fragmentierung).
- Lösung:
  - Der Speicher wird periodisch kompaktiert.

13.6.1 Einfache Speicherverwaltung: Probleme

- Vergessen Speicher anzufordern: Ref Variable erhält zufälligen Wert (evtl benutzte Speicher stelle
- Zuwenig Speicher angefordert: es wird benachbarter Speicher überschrieben, siehe oben
- Vergessen den Speicher freizugeben: Memory-Leak: Das Programm benötigt immer mehr Speicher. Bei virtuellem Speicher immer mehr auf Disk ausgelagert => langsamer, später Abbruch
- Der Speicher wird freigegeben obwohl noch verwendet: Dangling Pointer, Es wird auf eine anderweitig benutzte Speicherstelle zugegriffen und verändert => «komische» sich plötzlich verändernde Werte oft/meist Programmabbruch.

13.7 Automatische Speicherverwaltung

Hauptaufgabe der automatischen Speicherverwaltung ist die Freigabe des nicht mehr benötigten Speichers.

13.7.1 Einfache Algorithmen

Referenzzählung

- Es wird in jedem Objekt gezählt, wie viele Referenzen darauf verweisen.
- Wenn keine Referenz mehr darauf verweist ist, kann Objekt gelöscht werden.
- Operationen des Referenzzähler
  - Bei einer Zuweisung wird der Referenzzähler um 1 erhöht.
  - Bei Wegnahme einer Referenz wird der Referenzzähler um 1 erniedrigt.

Einfach, geringer Verwaltungsaufwand, Speicher wird frühestmöglich freigegeben, Fehler möglich durch Programmierer, Zusätzliche Operationen bei jeder Pointer-Zuweisung, Zyklische Strukturen können nicht freigegeben werden

Smart-Pointer Objekt-Referenzen (Pointer) sind nicht einfach «dumme» Adressen sondern «smarte» Objekte mit Referenz-Zähler.
Smart-Pointer merken selber: Wenn ihnen ein neuer Wert zugewiesen wird. Wenn Sie nicht mehr zugreifbar sind (out-of-scope gehen)
Vorteil: keine Fehler beim Erhöhen und Erniedrigen des Referenz-Zählers.
Nachteile: wie Referenzzählung, zyklischen Datenstrukturen werden nicht erkannt

13.7.2 Vollautomatische Algorithmen für Garbage Collection (GC)

Vollautomatisch heisst: System kann selbständig feststellen, ob Speicher noch benötigt wird.

Mark-Sweep-GC Speicher wird nicht sofort freigegeben sondern erst bei «Bedarf». Suche nach Blöcken, die freigegeben werden können, in zwei Phasen

1. MARK: Traversierung aller erreichbarer Objekte. Alle erreichbaren Objekte werden markiert
2. SWEEP: Sequentiell durch den Heap gehend, Speicher aller nicht markierten Objekte freigeben. Die Markierung von markierten Objekten wird gelöscht

Keine zusätzlichen Operationen bei Pointer-Zuweisungen nötig. Zyklische Datenstrukturen können aufgelöst werden. Aber: Grosser Aufwand. Das Programm muss währender Mark-Sweep-Phase gestoppt werden (Stop-The-WorldMechanismus). Es entstehen Löcher (Memory-Fragmentierung (Wenn Lücken im Heap zu klein um gefüllt zu werden))

Mark-Compact-GC Der Mark-Compact-Algorithmus hat den gleichen Markierungsprozess wie der Mark-Sweep-Algorithmus.
Dieser Algorithmus bereinigt jedoch nicht direkt die Objekte, die im Garbage gesammelt werden können. Stattdessen verschiebt er alle referenzierten Objekte an den Anfang des Heap, sodass keine Memory-Fragmentierung auftritt.
Vorteil: Analog Mark-Sweep-Algorithmus. Keine Memory-Fragmentierung.
Aber: Analog Mark-Sweep-Algorithmus. Aufwand ist noch grösser → schlechte Performance.

Copying-GC

- Der Speicher wird in zwei gleiche Teile (Semi Spaces) aufgeteilt:
  - Der eine Semi-Space enthält die aktuellen Objekte.
  - Der andere Semi-Space enthält ausschliesslich obsolete Objekte.
- Neue Daten werden im aktuellen Semi-Space angelegt.
- Wenn kein Platz mehr im aktuellen Semi-Space:
  - Es werden alle noch referenzierten Objekte in den anderen Semi-Space kopiert. Nicht mehr referenzierte Objekte bleiben liegen.
  - Die Rollen der Semi-Spaces werden vertauscht

Vorteil: es entstehen keine Löcher, die Suche nach freien Blöcken entfällt (belegter Bereich ist kompakt).
Nachteil: Es wird doppelt so viel Speicher benötigt. Aufwändiger Algorithmus. Es muss immer der ganze Speicher durchlaufen werden. Programm muss während dieser Zeit angehalten werden (Stop-The-World-Mechanismus).

Generational-GC Hypothese: Die meisten Objekte werden nur kurz gebraucht (z.B. das IteratorObjekt).
Idee: Die Objekte in (zwei, drei oder mehr) Generationen unterteilen. Neue Generationen werden häufiger nach freizugebenen Objekten durchsucht.

- Es handelt sich dabei nicht wirklich um einen neuen Algorithmus. Es können jetzt aber verschiedene Algorithmen zu unterschiedlichen Zeitpunkten je Generationstyp angewendet werden:
  - In der jungen Generation werden viele nicht referenzierte Objekte erwartet, der Coping-GC ist hier eine gute Lösung. Hierbei können die Objekte auch vom Bereich der jungen in den Bereich der alten Objekte kopiert werden.
  - In der alten Generation kann z.B. der Mark-Compact-Algorithmus eine gute Lösung sein.

13.8 Tuning & Tools der Garbage Collection (GC)

13.8.1 Weak/Soft-References (GC in Java)

Anwendungsbeispiel von Weak/Soft-References:
Datenstrukturen, die Sammlungen (Hastable, List usw.) von anderen Objekten beinhalten, benötigen interne Referenzen auf diese Objekte. Diese werden von GC gefunden und traversiert.

- Problem:
  - Objekte in Sammlungen können nicht freigegeben werden, wenn keine anderen Referenzen mehr auf diese Objekte vorhanden sind - obwohl diese eventuell nicht mehr benötigt werden.
  - Sehr viele Referenzen müssen traversiert werden (Laufzeit).
- Lösung:
  - Einführung von Referenzen, die nicht traversiert werden: Weak-References.
  - Weak-Reference: Objekt wird gelöscht, wenn nur noch Weak-References auf dieses zeigen.
  - Soft-Reference: Objekt wird gelöscht, wenn nur noch Soft-References auf dieses zeigen und der Heap-Speicher knapp wird.
- Vorteil:
  - Objekte, die nicht mehr benötigt werden, oder die bei Bedarf nachgeladen werden können, können freigegeben werden.
- Nachteil
  - Problem mit Dangling-Pointer.
  - Falls das Objekt doch noch benötigt wird, muss dieses wieder erstellt werden (nachladen)

13.8.2 Tuning (GC in Java

Mit folgenden Einstellungen kann der Garbage-Collector beeinflusst werden. Die Punkte 4 und 5 sollten nur im Ausnahmefall eingesetzt werden (Risiko von Artefakten).

1. Festlegung der Grösse der vier Speicherbereiche im Heap.
2. Auswahl eines bestimmten Garbage-Collectors (Java HotSpot bietet 4 Varianten, es gibt noch weitere). Die VM wählt diesen normalerweise selbst aus:
   (a) Serial GC: Single Thread, Generationen, Mark-Compact GC (Flag –XX:+UseSerialGC). Eignet sich z.B. für Microservices und Container mit wenig Speicher (100 MB).
   (b) Parallel GC: Analog Serial GC aber mit multiplen Threads (Flag -XX:+UseParallelGC). Falls Antwortzeit und Latenzzeit keine Rolle spielt, z.B. für Batchprozesse, oder falls Pausen von z.B. einer Sekunde akzeptabel sind.
   (c) Garbage First (G1) GC 1N) (Empfohlen und in der Regel Default): Generationen, inkrementell, parallel, hauptsächlich concurrent (Flag -XX:+UseG1GC). Kompromiss zwischen Performance und Latenzzeiten.
   (d) Z Garbage Collector (ZGC) 2N): Ähnlich G1, keine Generationen, unterbricht die laufenden Threads höchstens wenige ms, allerdings auf Kosten eines gewissen Durchsatzes (Flag -XX:+UseZGC).
3. Expliziter Aufruf des Garbage-Collectors.
4. Verwendung von Weak/Soft-References.
5. Einsatz von Finalizer (deprecated).

13.8.3 Garbage-Collector-Tools

JConsole - das Werkzeug zur Java Systemüberwachung:

- jconsole.exe im <jdk>/bin Verzeichnis.
- Zur Anzeige von:
  1. Memory
  2. Threads
  3. Prozessor-Auslastung

14 Vorlesung 06