Algorithmen und Datenstruktur VO (Univie)

Theorie

Algorithmus

Ein Algorithmus ist eine Schrittweiße Vorschrift zur Berechnung gesuchter aus gegebenen Größen, in der jeder Schritt aus einer Anzahl eindeutiger ausführbarer Operationen und einer Angabe über den nächsten Schritt besteht.

Eigenschaften (siehe Skript Seite 10): Eingangswert/ Ausgabewerte, Eindeutigkeit, Endlichkeit, Vollständigkeit, Korrektheit, Granularität der Operationen.

Greedy Algorithmus

- „gefräßiger, gieriger“ Ansatz
- Wahl des lokalen Optimums
- Effizienter Problemlösungsweg
- Oft sehr schnell
- Gute Lösungen
- Findet aber oft keine optimale Lösung!

Funktionsweiße

In jedem Schritt des Algorithmus wird die Möglichkeit gewählt, die unmittelbar den optimalen Wert bezüglich der Zielfunktion liefert. (Globale Sicht auf Endziel wird vernachlässigt)

Wenn man z.b. mit Greedy Münzen wechseln will (z.b. 18 Euro in 10 Euro, 6 Euro und 1 Euro Münzen), und möglichst wenige Münzen haben will, dann würde er als Lösung 1*10, 1*6 und 2*1 sagen (würde 18-10 rechnen, restliches Geld (8)- 10 geht nicht, deswegen mit 6 probieren, geht, also den Rest 8-6, dann mit 6 probieren, geht nicht, also mit 1 probieren, dass dann 2 mal geht). Aber optimale Lösung wäre 3*6, da 3 Mal die 6 Euro Münze genau 18 ergibt, und man somit eine Münze (4 bei Greedy gegenüber 3 beim optimalen) hätte

Divide and Conquer

Ausgehend von einer generellen Abstraktion wird das Problem iterativ verfeinert, bis Lösungen für verfeinerte Teilprobleme gefunden wurden, aus welchen eine Gesamtlösung konstruiert werden kann

3 verschiedene Ansätze:

- Problem size division (Skript Seite 51): Zerlegung eines Problems der Größe n in eine endliche Anzahl von Teilproblemen kleiner n (z.b Quicksort.)
- Step division (Skript Seite 52): Aufteilen einer Aufgabe in eine Sequenz (Folge) von individuellen Teilaufgaben

- Case division (Skript Seite 53): Identifikation von Spezialfällen zu einem generellen Problem ab einer gewissen Abstraktionsstufe

Dynamic Programing

Start mit Lösungen für Problemgröße 1, Kombination der berechneten Teillösungen bis eine Lösung für Problemgröße n erreicht wurde. Siehe Bsp Seite 55f.

Algorithmische Lücke

Eine algorithmische Lücke für ein Problem existiert, falls für die bekannten problemlösenden Algorithmen A gilt, dass ihr Aufwand größer als der ableitbare Lösungsaufwand für das Problem P ist. $\Omega(P) \leq 0(A)$

Master Theorem

- Ist ein „Kochrezept“ zur Bestimmung des Laufzeitverhaltens
- Vereinfachte Form (generelle Version Cormen et al. pp. 62)

Funktionsweiße

Es seien a ≥ 1, b ≥ 1 und c ≥ 0 Konstante
T(n) ist definiert durch aT(n/b) + Θ(n^c) wobei n/b entweder ⌊n/b⌋ oder ⌈n/b⌉ ist
T(n) besitzt dann den folgenden asymptotischen Grenzwert:

- Fall 1: wenn c < $log_b a$ dann T(n) = Θ($n^{log_b a}$)
- Fall 2: wenn c = $log_b a$ dann T(n) = Θ(n^c log n)
- Fall 3: wenn c > $log_b a$ dann T(n) = Θ(n^c)

Big Ω Notation

- Eine Funktion f(n) heißt von der Ordnung Ω(g(n)), wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \geq c_0 g(n)$ für alle n > n_0.
- Die Big-Ω-Notation liefert eine Untergrenze für die Wachstumsrate von Funktionen $f \in O(g)$ wenn f mindestens so schnell wie g wächst.
- z.B.: $2^n \in \Omega(n^2), n^{37} \in \Omega(n^2)$

Big Θ Notation

- Das Laufzeitverhalten eines Algorithmus ist Θ(n) falls gilt: $O(n) = \Omega(n)$
- Über Θ(n) kann das Laufzeitverhalten exakt beschrieben werden

Big O Notation

- Eine Funktion f(n) heißt von der Ordnung O(g(n)), wenn zwei Konstanten c_0 und n_0 existieren, sodass $f(n) \leq c_0 g(n)$ für alle n > n_0.
- Die Big-O-Notation liefert eine Obergrenze für die Wachstumsrate von Funktionen
- $f \in O(g)$ wenn f höchstens so schnell wie g wächst.
- z.B.: $17n^2 \in O(n^2), 17n^2 \in O(2^n)$

Strukturkomplexität

- Ziel: Metrik (Maßzahl) zu finden die den Aufbau, Struktur, Stil... eines Programmstücks bewerten und vergleichen lässt.
- Anforderungen: Gültigkeit (keine „Scheinergebnisse“), Einfachheit (Resultate verständlich + interpretierbar), Sensitivität (reagiert ausreichend auf unterschiedliche Ausprägungen), Robustheit („auslassen“ von uninteressanten Eigenschaften).
- Intra- modulare Komplexität: Beschreibt die Komplexität eines einzelnen Moduls
    - Modul Komplexität (intern)
        - LoC: Line- of- Codes
        - NCSS: Non commenting Source Statements (undokumentierter Code)
        - McCabe: zyklomatische Komplexität
    - Kohäsion (extern): Henry- Kafura Metrik (Informationsfluss)
- Inter- modulare Komplexität: Beschreibt Komplexität zwischen Moduln.

Kupplung vs. Kohäsion

- Kupplung: Misst Komplexität der Beziehungen zwischen Moduln.
- Kohäsion: Misst Informationsfluss von und nach Außen abhängig von der Modulgröße.
- Beziehung zwischen Kupplung und Kohäsion: starke Kupplung=> schwache Kohäsion, schwache Kupplung=> starke Kohäsion.
- Maß zur Bestimmung der Kohäsion: Beschreibt die funktionale Stärke des Moduls, zu welchem Grad die Modulkomponenten dieselben Aufgaben erfüllen.

Metrik von McCabe

- Ist ein Maß zur Beurteilung der Modul Komplexität
- Basiert auf zyklomatische Komplexität. V liefert die Anzahl der unabhängigen Pfade in einem Programmgraph. V > 5 ist einfach, 5- 10 ist normal, V > 10 komplex und V > 20 schwer verständlich.

Henry-Kafura Metrik

- Von S. Henry und D. Kafura
- Basiert auf Zusammenhang zwischen Modulkomplexität und Verbindungskomplexität zwischen Moduln.
- Zählt die Datenflüsse zwischen Moduln:
    - FAN IN: „Anzahl der Module die m verwenden“ bzw Anzahl der Datenflüsse, die im Modul m terminieren + Anzahl der Datenstrukturen aus denen das Modul m Daten ausliest
        - Wiederverwendbarkeit wird durch hohen FAN Wert „bestraft“
    - FAN OUT: „Anzahl der Module die m verwendet“ bzw Anzahl der Datenflüsse, die vom Modul m ausgehen + Anzahl der Datenstrukturen, die das Modul m verändert.
- Modulkomplexität (z.b LoC, McCabe) $C_{im} * (FAN\ IN_m * FAN\ OUT_m)^2$
- Annahme: Zusammenhang zwischen Komplexität und Fehlerkorrektur

Fenton und Melton Metrik

- Maß zur Bestimmung der Kupplung
- Kupplung misst die Unabhängigkeit zwischen Moduln
- Fenton Maß für die Kupplung zwischen 2 Moduln: $c(x,y) = i + \frac{n}{(n+1)}$ wobei i der schlechteste Kupplungstyp zwischen Modul x und y ist und n die Anzahl der „Kupplungen“ vom Typ i ist.

Paradigmen

Das was den Mitgliedern einer wissenschaftlichen Gemeinschaft gemein ist. Eine Konstellation von Meinungen, Wertungen und Methoden.

Algorithmen können auf unterschiedliche Art konzipiert sein: prozedural vs. funktional basiert vs. Logik basiert bzw orthogonal vs. objektorientiert.

Das logikbasierte Paradigma

Programm besteht aus Regeln und Fakten (siehe Bsp Skriptseite 12)

Das funktionale Paradigma

Das Programm ist nur aus Funktionen im mathematischen Sinne aufgebaut (Bsp Cäsar Verschlüsselung)

Jedes Vorkommen eines Funktionsaufrufes kann durch das Funktionsergebnis ersetzt werden.
3!>3*(3-1)!>3*2!....>6

Das prozedurale Paradigma

Der Lösungsweg ist durch eine Folge von Anweisungen vorgegeben. Anweisungen können Werte in Variablen zwischenspeichern, lesen und verändern. Z.b. Pfadfinder Geländespiel

Liste (Kapitel 3)

Speichertypen:

- Contiguous Memory:
  - Physisch zusammenhängender Speicherplatzbereich, äußerst starr, da beim Anlegen die endgültige Größe fixiert wird.
  - Verwaltung über das System
  - Datenstruktur auf der Basis von contiguous memory können nur eine begrenzte Anzahl von Elementen aufnehmen
- Scattered (Linked) Memory:
  - Physisch verteilter Speicherbereich, sehr flexibel, da die Ausdehnung dynamisch angepasst werden kann.
  - Verwaltung über das Programm

- Datenstrukturen können beliebig groß werden.

Stack

- Auf deutsch Kellerspeicher
- Ist ein Spezialfall der Liste die die Elemente nach dem LIFO (last in, first out) Prinzip verwaltet
- Stackoperationen können durch Listenoperationen ausgedrückt werden
- Man kann nur auf das oberste, zuletzt draufgelegtes Element zugreifen

Lineare Liste

- Beliebig große Anzahl an Elementen eines einheitlichen Typs
- Zugriff nur vom Kopf aus möglich
- Ende der Liste wird als Tail bezeichnet

Queue

- Warteschlange
- Ist ein Spezialfall der Liste die die Elemente nach dem FIFO (first in, first out) Prinzip verwaltet
- Elemente werden hintereinander angereiht, wobei nur am Ende der Liste Elemente angefügt und vom Anfang der Liste weggenommen werden können
- Anwendungen: Warteschlangen, Stoffwechsel, Bufferverwaltung
- Können durch Listenoperationen ausgedrückt werden

Spezielle Listen

- Doubly Linked List
- Circular List
- Ordered List
- Double Ended List

Informeller Vergleich

Datenstruktur	Stärken	Schwächen
Liste	Dynamisch Beliebige Datenmenge Klares Modell Geringer Speicherplatz	Linearer Aufwand der Operationen Simples Modell
Baum	Dynamisch Beliebige Datenmenge Logarithmischer Aufwand der Operationen	Balanzierungsalgorithmen aufwendiger Speicherplatzverbrauch Komplexes Modell
Vektor	Dynamisch Direkter Elementzugriff Konstanter Aufwand der Operation	Oft fixe Datenmenge Eingeschränkte Operationen

Baum

Eigenschaften

- stellt eine Generalisierung der Liste auf eine 2 dimensionale Datenstruktur dar
- besteht aus Menge von Knoten, die durch gerichtet Kanten verbunden sind
- ist ein spezieller Graph, der eine hierarchische Struktur über eine Menge von Objekten definiert
- nicht lineare Datenstruktur
- Exponentieller Zusammenhang zwischen Tiefe des Baumes und Anzahl der Knoten
- Zusammenhang zwischen Knoten(k)/ Blättern (b) und Höhe(h): 0(n) k/b: 0(log n)
- Stärken: dynamisch, beliebige Datenmenge, logarithmischer Aufwand der Operation
- Schwächen: Balanzierungsalgorithmen, aufwendigerer Speicherplatzverbrauch, komplexes Modell
- Laufzeit immer log n
- Anwendungen: allgemeine Schlüsselverwaltung, Haupt und Externspeichermanagement
- Außerhalb Informatik: Repräsentatives Wissen, Darstellung von Strukturen, Zugriffspfade,…

Aufbau

- Wurzel: einziger Knoten mit nur wegführenden Kanten
- Kante: Verbindung zwischen 2 Knoten
- (Interne) Knoten: Knoten in die Kanten hinein und von denen Kanten wegführen
- Elternknoten: Knoten der (dem) Knoten (Kindknoten) übergeordnet ist
- Kindknoten: Knoten der (dem) Knoten (Elternknoten) untergeordnet ist
- Transitive Eltern: Vorgänger vom Elternknoten
- Transitive Kinder: Nachfolger vom Kindknoten
- Blatt: Knoten von denen keine Kanten wegführen
- Pfad: Weg von der Wurzel zum Blatt (nur erlaubt: Pfad von oben nach unten!)
- Nachbarn/ adjazent: räumlich nebeneinander liegende Knoten auf derselben Tiefe

Längen

- Länge: Wege zwischen 2 Knoten entspricht der Anzahl der Kanten auf dem Weg zwischen den beiden Knoten
- Höhe: Länge des längsten Weges von diesem Knoten zu den erreichbaren Blättern
- Tiefe: Länge des Weges von dem Knoten zur Wurzel
- Höhe des Baumes: entspricht Höhe der Wurzel

Binärer Baum (BT)

Eigenschaften

- Knoten haben maximal 2 Nachfolger (Kinder)
- Sind geordnete Bäume

Arten

- Leerer Binärer Baum: knotenlos
- Voller Binärer Baum (VBB): Jeder Knoten hat keine oder 2 Kinder (kein Knoten besitzt nur 1 Kind!)
- Perfekter Binärer Baum (PBB): VBB und alle Blätter mit der selben Tiefe
- Kompletter Binärer Baum: PBB und Blattebene nicht vollständig, aber von links nach rechts gefüllt (rechts „fehlt“ z.b. Knoten)
- Höhen Balanzierter Binärer Baum: Für jeden Knoten ist der Unterschied der Höhe zwischen des linken und rechten Kindes maximal 1
- Expression Tree: Systematische Auswertung eines mathematischen Ausdrucks wobei Blätter Operanden repräsentieren und Knoten Operatoren darstellen

Traversierungsalgorithmus

- Traversieren**:** Bezeichnet das systematische besuchen bei einem Baum all seiner Knoten
- Bearbeiten eines Knoten
- Anschließendes rekursives besuchen und bearbeiten des linken Kindes
- Anschließendes rekursives besuchen und bearbeiten des rechten Kindes
- 3 Bearbeitungsreihenfolgen interessant:
  - Preorder Traversierung: Wurzel=> Links=> Rechts
  - Postorder Traversierung: Links=> Rechts=> Wurzel
  - Inorder Traversierung: Links => Wurzel => Rechts („von links nach rechts gehen und von unten nach oben“))

Binärer Suchbaum (BST)

- Jeder (interne) Knoten besitzt eine linke und rechte Verbindung, die auf einen Binärbaum oder einen externen Knoten verweist, wobei links <= gilt und rechts >
- Nullverbindungen: Verbindungen zu externen Knoten
- Externen Knoten: besitzen keine weiteren Verbindungen oder Schlüssel
- BST ist ein BT bei dem jeder interne Knoten einen Schlüssel besitzt
- Aufwand proportional zur Höhe des Baumes und nicht zu Anzahl des Elemente
- Einfügen, Zugriffe und Löschen im Durchschnitt von 0(log n)
- Zugriff auf Elemente in sortierter Reihenfolge durch inorder Traversierung

Funktionsweiße

- Suche nach Schlüssel: Pfad wird von Wurzel abwärts verfolgt
  - Bei jedem internen Knoten wird der Schlüssel x mit dem Suchschlüssel s verglichen
  - Falls x = s liegt ein Treffe vor, falls s <= x suche im linken Teilbaum, sonst im rechten
  - Wenn man externen Knoten erreicht, war die Suche erfolglos
- Einfügen entspricht einer erfolglosen Suche und dem Anfügen eines neuen Knotens an der Nullverbindung wo die Such endet (anstelle des externen Knotens)
- Löschen unterscheidet 3 Fälle

- Keinem internen Knoten als Kind
  - Anhängen des verbleibenden Teilbaums an den Elternknoten des zu löschenden Knoten => Knoten löschen und Nullverbindung setzen
- Einem internen Knoten als Kind
  - Anhängen des verbleibenden Teilbaums an den Elternknoten des zu löschenden Knoten => Knoten löschen und nächsten „einhängen“
- Zwei internen Knoten als Kind
  - Geeigneter Ersatzknoten muss gefunden werden: entweder kleinste im rechten oder größte im linken Teilbaum

Höhenbalanzierter Binärer Suchbaum

- Für jeden Knoten ist der Unterschied der Höhe zwischen des linken und rechten Kindes maximal 1
- Vermeiden von Entartung der Laufzeit zu 0 (n) und garantiert 0 (log n)
- ist höchstens 44,04 % höher als ein vollständig balancierter binärer Suchbaum

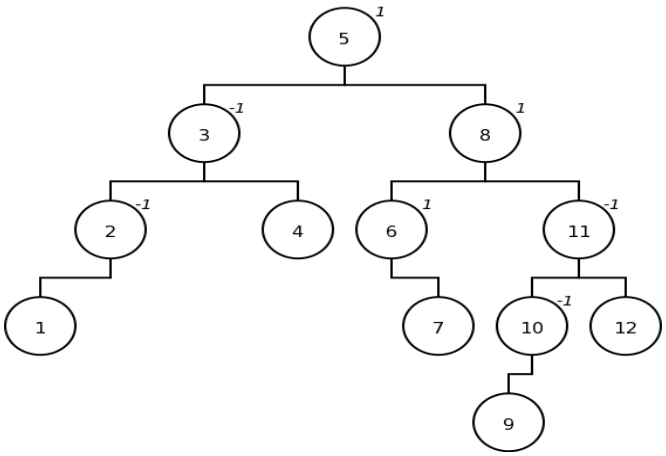
AVL Baum

- Ist ein höhenbalanzierter binärer Suchbaum
- Von **A**delson- **V**elski und **L**andau 1962 entwickelt
- Anmerkung: externe Knoten nicht vergessen!

Funktionsweiße

- Einfügen: Verlauft analog zum Einfügen im binären Suchbaum, wobei die Balanzierungseigenschaft verletzt werden kann=> Korrektur notwendig!
  - Knotenrotation:
    - Einfache: CA(l)B(l)=>BAC
    - Doppelte: Einfache erzeugen, dann Rotation: CA(l)B(r)=> BAC
  - LL und RR Rotation
    - LL Fall: B3(r)A(l)1(l)2(r)=>A1(l)B(r)2(l)3(r)
    - RR Fall: A1(l)B(r)2(l)3(r)=> B3(r)A(l)a(l)2(r)
- Löschen: Verlauft analog zum Löschen im binären Suchbaum, wobei die Balanzierungseigenschaft verletzt werden kann=> Korrektur notwendig!
  - Lösung: Rotation analog zum Einfügen, kann aber dadurch wieder Balanzierung eines anderen Knoten verletzen. Kann also Serie von Rotationen auslösen.

Beispiel eines AVL Baums:



Mehrwegbaum

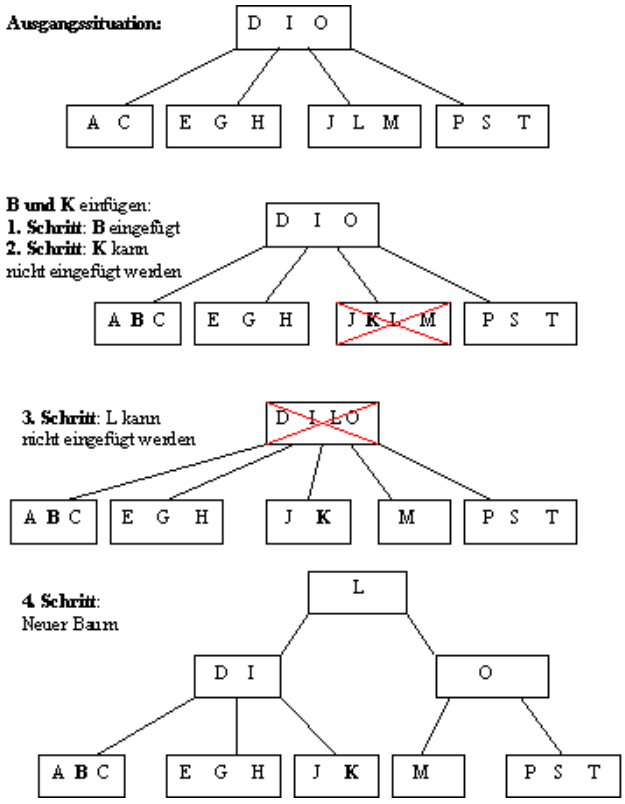
- Bäume mit Knoten, die mehr als 2 Kinder besitzen können
- Intervallbasierte Suchdatenstruktur

Funktionsweiße

- Suche: Ähnlich der Suche in BST
    - Falls man externen Knoten erreicht: Suche erfolglos

2-3-4 Baum

- Ist ein Mehrwegbaum (mit 2 Zusatzeigenschaften)
- Jeder Knoten hat mindestens 2 und maximal 4 Kinder (Größeneigenschaft)
- Alle externen Knoten besitzen dieselbe Tiefe (Tiefeneigenschaft)



Höhenbalanzierter Mehrwegbaum (B+- Baum)

Eigenschaften

- Balanzierter Mehrwegbaum, Externspeicherverwaltung, Datenbanksysteme
- Externspeicher Datenstruktur
- Eine der häufigsten Datenstrukturen in Datenbanksystemen
- Dynamische Datenstruktur
- Algorithmen für Einfügen und Löschen erhalten die Balanzierungseigenschaft
- Garantiert einen begrenzten (worst case) Aufwand für Zugriff, Einfügen und Löschen

- Maximal Ordnung log n (0(log n)) für Zugriff, Einfügen, Löschen
- Besteh aus Index und Daten
- Der Weg zu allen Daten ist gleich lang
- Maximal 45% „zu groß“ (Seite 233 im Skript)

Aufbau

- Alle Blattknoten haben die gleiche Tiefe (gleiche Weglänge zur Wurzel)
- Wurzel ist entweder ein Blatt oder hat mindestens 2 Kinder
- Jeder (interne) Knoten besitzt mindestens k und maximal 2k Schlüsselwerte, daher mindestens k+1 und maximal 2k+1 Kinder (Ausnahme Wurzel)
- Unterbaum: die in ihm gespeicherten Elemente sind kleiner als die Elemente im rechten und größer als im die Elemente im linken Unterbaum.
- Die Intervallgrenzen werden durch die Schlüsselwerte bestimmt

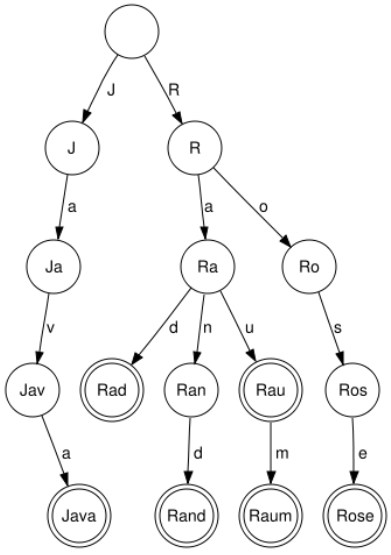
Funktionsweiße

- Suche: Bereichsabfrage möglich: Suche alle Elemente im Bereich [Untergrenze, Obergrenze]
- Einfügen:
    - Platz vorhanden: Einfach an freier, passender Stelle einfügen
    - Platz nicht vorhanden: Überlauf=> Externer Knoten muss geteilt werden
    - Kein Platz in Knoten und Indexknoten=> Indexknoten muss gesplittet werden
        - Mittlere Indexelement in darüber liegenden Indexknoten eintragen, und falls der nicht existiert, wird eine neue Wurzel erzeugt

Trie

- Preifx- Baum, Zeichenkettenverwaltung, Wörterbuch
- Digitaler Suchbaum
- Dient zur Speicherung von Strings
- Bei k Buchstaben ein „k+1- ary Tree“, wobei jeder Knoten durch eine Tabelle mit k+1 Kanten auf Kinder repräsentiert wird
- Spezialformen: Patricia Tree, de la Braindais Tree

Bsp für einen Tree:



Patricia Tree: Ziel ist die Komprimierung des Tries
de la Braindais Tree: Listen Repräsentation eines Tries. Statt der Tabellen im Knoten lineare Liste. Knoten besteht aus 2 Kinderkomponenten. Nächster Wert- nächster Level

Sortierverfahren

Insitu: Sortierverfahren das mit ursprünglichen Datenbereich auskommt
Exsitu: Sortierverfahren das mit ursprünglichen Datenbereich nicht auskommt=> zusätzlicher Speicherplatz wird benötigt
Stable vs. Unstable: Bei einem stabilen Sortierverfahren wird die relative Reihenfolge von gleichen Schlüsseln nicht verändert. Stabilität garantiert Korrektheit des Algorithmus.

Elementare/ Simple Verfahren: Selection Sort, Bubble Sort
Verfeinerte (Intelligentere) Verfahren: Quicksort, Mergesort

Selection Sort

- Wird auch Minimumsuche genannt
- Findet kleinstes Element, vertauscht es mit dem Element an nächster Stelle und wiederholt diesen Vorgang für alle noch nicht sortierten Elemente.
- Hauptspeicheralgorithmus
- Insitu Algorithmus
- Sortiert im eigenen Datenbereich
- Braucht nur temporäre Variable
- Konstanter Speicherplatzverbrauch
- Aufwand generell 0 (n^2)
- Stabiles Sortierverfahren

Beispiel mit 0405144
0405144 //0 passt=> lassen
0405144 //tausch 40
0045144 //tausch 41
0015444 //tausch 54
0014544 //tausch 54
0014454 //tausch 54
0014445 //5 passt=> fertig

Beispiel mit 0401099
0401099 //0 passt=> lassen
0401099 //tausch 40
0041099 //tausch 40
0001499 //1 passt=> lassen
0001499 //4 passt=> lassen
0001499 //9 passt=> lassen
0001499 //9 passt=> fertig

Beispiel mit 0208189
0208189 //0 passt=> lassen

0208189 / tausch 20
0028189 //tausch 21
0018289 //tausch 82
0012889 //9 passt=> fertig

Beispiel mit 0402913
0402913
0042913
0012943
0012349

Beispiel mit 0402196
0402196
0042196
0012496
0012469

Bubble Sort

- Wenn 2 benachbarte Elemente aus der Ordnung sind, vertausche sie. Dadurch wird beim ersten Durchlauf das größte Elemente an die letzte Stelle gesetzt, beim 2ten Durchlauf das 2. größte Elemente and die vorletzte Stelle, usw. Elemente steigen wie Blasen auf
- Hauptspeicheralgorithmus
- Insitu Algorithmus
- Sortiert im eigenen Datenbereich
- Braucht nur temporäre Variable
- Konstanter Speicherplatzverbrauch
- Aufwand generell 0 (n^2)
- Stabiles Sortierverfahren

Beispiel mit 0402196 (so sollte es laut Wikipedia richtig sein)
0042196 //4 größer als 2=> tauschen
0024196 //4 größer als 1=> tauschen
0021496 //9 größer als 6=> tauschen
0021469// 2 größer als 1=> tauschen
0012469// alles passt=> fertig

Beispiel mit 0401099 (ab hier fehlen die „Zwischenschritte“)
0401099 //4-0, 4-1, 4-0
0010499 //1-0
0001499

Beispiel mit 0405144
0405144
0041445
0014445

Beispiel mit 0402913
0402913

0024139
0021349
0012349

Beispiel mit 0208189
0028189
0021889
0012889

Quicksort

- Entwickelt 1962 von Hoare
- „Vektor“ wird in 2 Teile geteilt, sodass alle Elemente links vom Pivotelement kleiner und alle Elemente rechts größer sind (divide and conquer Eigenschaft).
- Die Teilvektoren werden unabhängig voneinander wiederum mit Quicksort sortiert
- Hauptspeicheralgorithmus
- Insitu Algorithmus
- Sortiert im eigenen Datenbereich
- Braucht nur temporäre Variable
- Konstanter Speicherplatzverbrauch
- Aufwand im Durchschnitt 0(n*log(n)), kann aber zu $0(n^2)$ entarten
- Empfindlich auf unvorsichtige Wahl des Pivotelements
- Stabilität nicht einfach realisierbar
- Rekursiver Algorithmus

Funktionsweiße

- Beliebiges Element als Pivotelement wählen
- Pivotelement von links bzw recht in den Vektor „Hineinsinken“ lassen (sequentiell von links mit allen kleineren Elementen, von rechts mit allen größeren vertauschen)
- Wenn links kleineres und rechts größeres Element ist, diese beide tauschen.

Beispiel mit 0401099 (Pivot beliebig auswählen aber in Regel erstes Element)
0401099 (0 von links auf 4 und von rechts auf 0) 0 Fixpunkt gefunden
401099 (4 von links nach 9 und von rechts auf 0) 4 Fixpunkt gefunden
010 99 (0 von links auf 1 und von rechts auf 0) 0 Fixpunkt und 99 ebenfalls
10 (1 von links auf 0 und von rechts auf 1) => Tauschen
01 (passt)=> Fertig

Beispiel mit 0405144 (von links: größeres suchen, von rechts: kleineres suchen)
0405144 (0 von links auf 4 und von rechts auf 0) Fixpunkt gefunden
405144 (4 von links auf 5 und von rechts auf 1=> 5 und 1 tauschen)
401544 (4 von links auf 5 und von rechts auf 1) Fixpunkt gefunden
01 4 544 (01 fertig, genauso wie 4)
544 (5 von links auf ende und von links auf anfang)
445 (passt)=> Fertig

Beispiel mit 251436
251436 (2 von links auf 5 und von rechts auf 1)=> 5 und 1 tauschen

215436 (2 von links auf 5 und von rechts auf 1)=> Überkreuzung=> 2 fix zwischen 1 und 5
1 5436 (1 fix)
5436 (5 von links auf 6 und von rechts auf 3)=> Überkreuzung=> 5 fix zwischen 3 und 6
43 6 (6 fix)
43 (4 von links auf 3 und von rechts auf 4)=> tauschen
34 (passt)=> Fertig

Mergesort

- Entwickelt durch John von Neumann 1938
- Divide and onquer Ansatz
- Hauptspeicheralgorithmus aber auch als Externspeicheralgorithmus verwendbar
- Exsitu Algorithmus
- Braucht einen zweiten ebenso großen Hilfsvektor zum Umspeichern
- Aufwand generell 0(n*log(n))
- Braucht doppelten Speicherplatz
- Rekursiver Algorithmus
- stabiler Sortieralgorithmus

Funktionsweiße

- Der zu sortierende Vektor wird rekursiv in Teilvektoren halber Länge geteilt, bis die Vektoren aus einem einzigen Element bestehen
- Danach werden jeweils 2 Teilvektoren zu einem doppelt so großen Vektor gemerged (beim mergen werden sukzessive die ersten beiden Elemente der Vektoren verglichen und das kleinere in den neuen Vektor übertragen, bis alle Elemente im neunen Vektor sortiert gespeichert sind)
- Das Mergen wird so lange wiederholt, bis in der letzten Phase 2 Vektoren der Länge n/2 zu einem sortierten Vektor der Länge n verschmelzen.

Heapsort

- Entwickelt 1964 durch Williams
- Eine Heap (Priority Queue) kann Basis zum Sortieren bilden
- Kombination von beiden Arrays=> Vermeidung von doppeltem Speicherplatz
- Hat nicht Nachteil von Mergesort (doppelter Speicherplatz) und Quicksort (Entartung möglich), hat aber höheren konstanten Aufwand!
- Nicht stabil!

Funktionsweiße

- Ein Element nach dem anderen in einen Heap einfügen
- Sukzessive das kleinste bzw. das größte Element entfernen
- Die Elemente werden in aufsteigender bzw. absteigender Ordnung geliefert

Comparison Sort Verfahren

- Deutsch „Vergleichende Sortierverfahren“

- Reihenfolge der Elemente wird bestimmt indem Elemente miteinander verglichen werden.
- Günstige Grenze von 0(n log n)

Andere Sortierverfahren

- Sortierverfahren die nicht auf dem Vergleich zweier Werte beruhen, können eine Eingabe Sequenz in linearer Zeit sortieren.
- Erreichen die Ordnung 0(n)
- Bsp: Counting Sort, Radix Sort, Bucket Sort

Counting Sort

- Erstmals 1954 von Harold H. Seward (MIT) verwendet
- Bedingung: Elemente sind Integer Werte
- Laufzeit von 0(n) (wenn k Ordnung n ist=> (k=0(n)))
- Stabiles Sortierverfahren (da letzte Schleife von hinten nach vorne arbeitet)
- Infos auf deutsch unter http://de.wikipedia.org/wiki/Countingsort
- ist nicht insitu Verfahren, daher doppelter Speicherplatz notwendig

Funktionsweiße

- Jedes Element wird einfach auf die Position seines Wertes in einem Zielvektor gestellt
- Idee: Bestimme für jedes Element x die Anzahl der Elemente kleiner als x im Vektor, verwende diese Information um das Element x auf seine Position im Ergebnisvektor zu platzieren.

Beispiel mit 0401099
7 Stellen=> n = 7
Höchste Zahl: 9=> *k* = 9

3 1 0 0 1 0 0 0 0 2 (Häufigkeit der Zahl)

0 1 2 3 4 5 6 7 8 9 (Zahlenwert (0 bis k))

3 4 4 4 5 5 5 5 5 7 (Stelle im Ergebnis/ wenn man Stelle braucht, Stelle – 1 nachher!!!)

0 1 2 3 4 5 6 7 8 9 (Zahlenwert (0 bis k))

0 0 0 1 4 9 9 (Ergebnis)

Beispiel mit 0405144
7 Stellen=> n = 7
Höchste Zahl: 5=> k = 5

2 1 0 0 3 1 (Häufigkeit der Zahl)

0 1 2 3 4 5 (Zahlenwert)

2 3 3 3 6 7 (Stelle im Ergebnis/ wenn man Stelle braucht, Stelle – 1 nachher!!!)

0 1 2 3 4 5 (Zahlenwert)

0 0 1 4 4 4 5 (Ergebnis)

Beispiel mit 0402913
7 Stellen=> n = 7
Höchste Zahl: 9=> k = 9

2 1 1 1 1 0 0 0 0 1 (Häufigkeit der Zahl)

0 1 2 3 4 5 6 7 8 9 (Zahlenwert)

2 3 4 5 6 6 6 6 6 7 (Stelle im Ergebnis/ wenn man Stelle braucht, Stelle – 1 nachher!!!)

0 1 2 3 4 5 6 7 8 9 (Zahlenwert)

0 0 1 2 3 4 9 (Ergebnis)

Erweiterung

- Es gibt keine doppelten Werte

Radix Sort

- Betrachtet Struktur der Schlüsselwerte
- Prinzip lässt sich auf alphanumerische Strings erweitern.
- Aufwand: 0(n)

Funktionsweiße

- Schlüsselwerte der Länge b werden in einem Zahlensystem der Basis M dargestellt (M ist der Radix)
- Es werden Schlüssel sortiert, indem ihre einzelnen Bits an derselben Bit Position miteinander verglichen werden.

Bsp: Es sollen folgende Zahlen geordnet werden: 124, 523, 483, 128, 923, 584

Partitionierungsphase (Zunächst wird nach der letzten Stelle geordnet.):

|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
| | | |
523 124 128
483 584
923

Sammelphase (Elemente von links nach rechts, von oben nach unten aufsammeln): 523, 483, 923, 124, 584, 128

Erneute Partitionierungsphase (nun nach der mittleren Stelle (im Allgemeinen von rechts nach links jeweils eine Stelle weiter).

|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
| |

523 483
923 584
124
128

Nun eine zweite Sammelphase: 523, 923, 124, 128, 483, 584

Letzte Partitionierungsphase (jetzt wird nach der ersten Stelle geordnet):

|0| |1| |2| |3| |4| |5| |6| |7| |8| |9|
| | | | |
124 483 523 923
128 584

letzte Sammelphase (ist zugleich „Ergebnis“): 124, 128, 483, 523, 584, 923

LSD- Radixsort

- Radix Sort der von rechts nach links läuft
- Stabiles Sortierverfahren
- Wird auch Straight Radix Sort genannt

Funktionsweiße

- Idee: Betrachtet Stellen von rechts nach links
- Sortiere Datensätze stabil bezogen auf das betrachtete Bit.

Binärer Quicksort

- Radix Sort der von links nach rechts läuft
- Unterschied zu LSD- Radixsort ist das man die Datensätze in M unabhängige Gruppen gibt und rekursive sortiert
- Wird auch Radix Exchange Sort genannt
- Sehr altes Verfahren aus 1929
- Gut geeignet für kleine Zahlen
- Benötigt relativ wenig Speicher als andere Lineare Sortierverfahren
- Stabiles Sortierverfahren
- Ähnlich Quicksort nur für positive Zahlen

Funktionsweiße

- Idee: Betrachtet Stellen von links nach rechts
- Sortiere Datensätze bezogen auf das betrachtet Bit
- Teile die Datensätze in M unabhängige, der Größe nach geordnete, Gruppe und sortiere rekursive die M Gruppen wobei die schon betrachteten Bits ignoriert werden.
- Aufteilung der Datensätze durch insitu Ansatz ähnlich des Partitionierens beim Quicksort.

Bucket Sort

- Annahme über die n Eingabe Elemente, dass sie gleichmäßig über das Intervall [0,1] verteilt sind
- Ist eine Variante des Radixsort Verfahrens
- Aufwand: 0(n) (außer beim Suchen!)

Funktionsweiße

- Idee: Teile das Intervall [0,1] in n gleichgroße Teil Intervalle (Buckets) und verteile die n Eingabe Elemente auf die Buckets. Da die Elemente gleichmäßig verteilt sind, falle nur wenige Elemente in das gleiche Bucket.
- Sortiere die Elemente pro Buckets mit Selection Sort
- Besuche die Buckets entlang des Intervalls und gib die sortierten Elemente aus

Beispiel mit 0401099

Höchste Zahl: 9=> k=9 (nun Werte von 0 bis k (also 9) auflisten)
0->0->0->0
1->1
2
3
4->4
5
6
7
8
9->9->9

Man liest von oben nach unten die Zahlen „durch“ also 000 dann 1 dann 4 dann 99=>
Ergebnis ist 0001499

Externes Sortieren

- Alle Sortierverfahren, die nicht ausschließlich im Hauptspeicher ablaufen
- Benötigen sekundäre (Platten) oder tertiäre (Bänder) Speichermedien
- Grund: zu große Datenmenge für Hauptspeicher
- Ziel: Anzahl der Transfer zwischen Hauptspeicher und Externspeicher zu minimieren
- Verwaltung elementarweise oder blockweise

„Sortieren und Mischen“

- Seit 50er Jahre im Einsatz
- So viele Daten wie möglich in Hauptspeicher laden und sortieren
- Sortierte Daten (Run) auf externes Speichermedium schreiben
- Runs über den Hauptspeicher zu größeren Runs zusammenmischen (Merging)

Balanced Multiway Merging

- Idee: Anfänglicher Verteilungsdurchgang und mehrere Mehrweg Mischdurchgänge
- Annahme: N zu sortierende Datensätze auf externem Gerät, Platz für M Datensätze im Hauptspeicher, 2 P externe Geräte zur Verfügung
- Ziel: sortiertes Ergebnis auf Band 0
- Aufwand: log_p (N/M) Durchlaufe wobei N Anzahl Elemente ist und M Größe des Hauptspeichers

Funktionsweiße

- Bilden initaler Runs durch sortieren der Daten im Hauptspeicher und gleichmäßiger Verteilen auf P Output Bänder
- Alternieren der Bänder (Input wird Output,…)
- Merging der „Runs“ von den Input auf die Output Bänder und alternieren der Bänder bis ein sortierter Run entsteht.

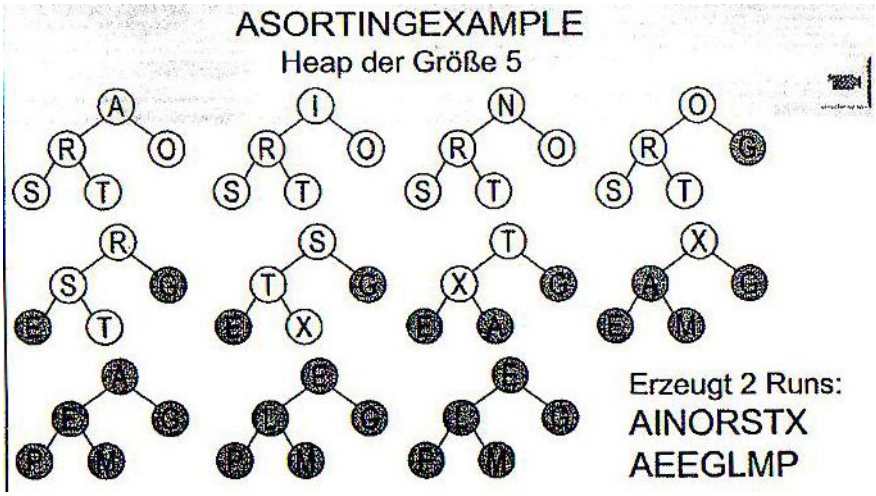
Beispiel mit ASORTINGANDMERGINGEXAMPLE*
Speicher 1: AOS* DMN* AEX* // auf 1 Band geben und jeweils in 3er Gruppe sortieren
Speicher 2: IRT* EGR* LMP*
Speicher 3: AGN* GIN* E*
Speicher 1: AAGINORST// sortierte 3er Gruppe zusammen nehmen und sortieren
Speicher 2: DEGGIMNNR
Speicher 3: AEELMPX
Ergebnis => AAADEEEGGGIILMMNNNOPRRSTX*//3 Bänder vereinen + sortieren

Replacement Selection

- Elemente im Hauptspeicher werden über die Priority Queue (PQ) der Größe p verwaltet (sortiert)
- Es lässt sich zeigen, dass Runs die mit Replacement Selection erzeugt wurden ungefähr 2 Mal so groß sind wie die mit Priority Queue (PQ)
- Nachteil: große Anzahl an Bändern=> deswegen auch „Entwicklung“ von Polyphase Merging

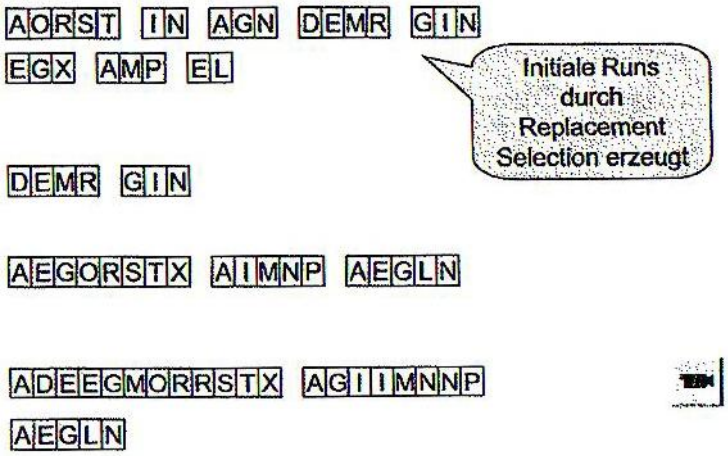
Funktionsweiße

- PQ wir mit den kleinsten Elementen der p Runs gefüllt
- Das kleinste Element der PQ wird auf das Output Band geschrieben und das nächste Element nachgeschoben.
- Die PQ Eigenschaften wird mit einer heapify Funktion erhalten
- Im Hauptspeicher sind nur die kleinsten Elemente der Runs, Realisierung über Pointer auf die echten Runs=> indirekte Heap
- Idee ist den ungeordneten Input durch eine große PQ durchzuschleusen
- Das kleinste Element Rauschreiben und das nächste Element in die PQ aufzunehmen
- Spezielle Situation falls ein Element kleiner als das letzte geschrieben ist (kann nicht mehr Teil des Runs werden, wird markiert und als größer als alle Elemente des aktuellen Runs behandelt)
- Diese Element beginnt einen neuen Run



Polyphasen Merging

- Nachteile des Balanced Multiway Merging ist die relativ große Anzahl von benötigten Bändern und Ansatz mit P Bändern und 1 Output Band benötigt zwar weniger Bänder, aber exzessives Kopieren
- Idee von Polyphasen Merging: Verteile die mit Replacement Selection erzeigten initialen Runs ungleichmäßig über die Bänder (und ein Band bleibt leer)
- Führe ein Merging- until- empty durch, also Merging bis ein Band leer ist, welches das neue Output Band wird
- Diese Merging- until- empty Strategie kann auf beliebige Bandzahlen (>2) angewendet werden
- Es werden für n Wege merging nur n+1 Bänder benötigt
- Analyse ist kompliziert
    - Unterschied zwischen Balanced Multiway Sort und Polyphase Merging eher gering
    - Polyphase Merging nur für geringe Bandzahlen (p<9) besser als Balanced Multiway Sort
        - Dient eher zum verringern der Bandzahlen



Vektoren (Kapitel 5)

Dictionary

- Unterstützt nur löschen, einfügen und suchen
- Element besteht aus Schlüssel (key) und einem Informationsteil (info)
- Wird verwendet für Wörterbücher, Bestandsliste, Namensliste,…

Beispiel für eine „Lagerhaltung“:

Schlüssel	Info
1	CPU
2	Bildschirm
3	Tastatur
4	Maus
5	HD
6	RAM

Hashing

- Ist eine Methode Elemente in einer Tabelle direkt zu adressieren. Schlüsselwerte=> arithmetische Transformation=> Tabellenadresse (Indizes). Daher Index wird aus dem Schlüsselwert berechnet
- Bei Kollision kommt es zu einer Kollisionsbehandlung (wenn 2 oder mehrere Schlüsselwerte auf dieselbe Tabellenposition abgebildet werden)

Notiz zu mod: $a \; mod \; m = a - \left(\frac{a}{m}\right) * m$ (sprich der Restbetrag bei einer Division)

Beispiel:1 (CPU), 17 (Tastatur), 4 (TFT), 25 (HD) und Tabelle hat 7 Plätze=> mod 7

Mod Ergebnis	Eigentlicher Wert	Zu speichern (Begründung)
0		
1	1	CPU (da 1 mod 7 =>1)
2		
3	17	Tastatur (da 3 mod 7 =>3)
4	4	TFT (da 4 mod 7 =>4)
5		

Es kommt dann aber bei 4 zu einem Konflikt da 25 (HD) (4 mod 7 =>4) selben Platz hätte!

Bei 25 (HD) (4 mod 7 =>4) kommt es also zu einem Konflikt wegen der Kollision=> Kollisionsbehandlung.

Es muss also ein neuer, freier Platz gefunden werden, wofür es 2 Verfahren gibt:

- Separate Chaining
- Double Hashing

Separate Chaining

- Kollisionsbehandlung durch lineare Liste (z.b. Oben würde man. einfach HD auf eine neue Liste setzen, und bei 4 anhängen)
- Dynamische Datenstruktur, beliebig viele Elemente
- Speicherplatzaufwand sehr hoch

Double Hashing

- Es wird eine zweite, von h unabhängige Hashfunktion zur Bestimmung einer alternativen Position verwendet.
- Spezialfall ist Linear Probing

Beispiel: h (k) = k mod 7
g (k) letzte Ziffer von k mod 3

a0 = 25 mod 7 = 4 (Annahme: es kommt zur Kollision)
a1 = (4+(5*3)) mod 7 = 5

a0 = 39 mod 7 = 4(Annahme: es kommt zur Kollision)
a1 = (4+(9*3)) mod 7 = 3 (Annahme: es kommt zur Kollision)
a2 = (3+(9*3)) mod 7 = 2

Linear Probing

- Man verwendet den nächsten möglichen freien Platz
- Optimale Speicherplatzausnutzung
- Statische Datenstruktur, Tabellengröße vorgegeben
- Kollisionsbehandlung komplex
- Wieder frei werdende Positionen müssen wegen Kollisionsbehandlung als „wieder frei“ markiert werden

Beispiel kurz:
25 mod 7 = 4 (Annahme: es kommt zur Kollision) => 4+1 = 5=> kommt auf Platz 5
39 mod 7 = 4 => Kollision=> 4+1 = 5=> Kollision=> 5 + 1 = 6 => kommt auf Platz 6

Beispiel lang: MatrikelNummer + Geburtstagtag + Jahr + Regel dass wenn Zahl doppelt vorkommt, sie plus 10 gerechnet wird.
04051441424031986=> 0 4 10 5 1 14 24 2 34 20 3 11 9 8 6
k=> Zahl + 1

Rechnungen:
0 mod 17 = 0
24 mod 17 = 7
34 mod 17 = 0
1mod 17= 1 => Kollision
2 mod 17 = 2 => Kollision
3 mod 17 = 3
20 mod 17 = 3 => Kollision=> nächste freie Stelle ist 6

Ergebnis (Mod Ergebnis/ Wert der gespeichert wird):

0	0
1	1
2	2
3	23
4	4
5	5
6	20
7	24
8	9
9	10
10	11
11	8
12	16
13	14

Dynamische Hashing Verfahren

- Versuchen im Fall von Kollisionen die ursprüngliche Tabelle zu erweitern
- Linear Hashing, Extensible Hashing und Bounded Index Size sind Verfahren die darunter fallen

Beispiel:

Wert	k(x)	binär	h2(x)	h3(x)
7	7	00111	11=3	111=7
8	8	01000	00=0	000=0
9	9	01001	01=1	001=1

Linear Hashing

000	8			X	2 = 000010
01	17	25		X	8 = 001000
10	34	50	2	X	17 = 010001
11				X	25 = 011001
100	28			X	28 = 011100
					34 = 100010
					50 = 110010

mit Blockgröße b = 3, NtS = 1
und Rundennummer d = 2

Suche: a = hd (x)
If (a< NexttoSplit) a = hd +f(x)

Einfügen Beispiel (Weiteres Bsp siehe Skript Seite 313):
6 = 000110 in oberen Block
NexttoSplit (NtS) = 2, b = 3, d = 2

8			

000
001
10
11
100
101

17	25		
34	50	2	
28			

6 kommt dann in einen Überlaufblock

Damit erklärt sich auch das splitten. d wird erhöht wenn das Splitting einmal durch ist.

Allgemein: Bei Überlauf eines Blockes wird der Primär und der Überlaufbereich um 1 Block erweitert

Extensible Hashing

- Wenn Primärbereich zu klein=> Verdoppelung
- Primärbereich wird durch Index T verwaltet
- Bei Überlauf Split und Verdoppelung des Index
- Löschen leerer Blöcke schwierig
- Indexwachstum worst case

Beispiel: Einfügen von 6 = 00110
2 Fälle:

- t<d => mehrere Indexeinträge referenzieren diesen Block
- t = d => ein Indexeintrag referenziert diesen Block

t<d

00 Link von 00 und 10 zu Block mit 14, 8, 10 ,26 (Block 1)
01 Link von 01 zu Block mit 9, 13 (Block 2)
10 Link von 11 zu Block mit 7, 31, 23 (Block 3)
11

Globale Tiefe d = 2 Lokale Tiefe (t) bei Block 1 1, bei Block 2 und 3 jeweils 2

=>

00 Link von 00 zu Block mit 8 (Block 1)
01 Link von 10 zu Block mit 14, 6, 10, 26 (Block 2)
10 Link von 01 zu Block mit 9, 13 (Block 3)
11 Link von 11 zu Block mit 7, 31, 23, 15 (Block 4)

Lokale Tiefe (t) bei Block 1, 2, 3 und 4 jeweils 2

t=d: dazu noch 15 und 19 einfügen=> neue Referenzen anfordern, Index Verdoppelung
15 = 01111
19 = 10011
d = 3

000 Link von 000 und 100 zu Block mit 8 (Block 1)
001 Link von 001 und 101 zu Block mit 9, 13 (Block 2)
010 Link von 010 und 110 zu Block mit 4, 6, 10, 26 (Block 3)
011 Link von 011 und 111 zu Block mit 7, 31, 23, 15 (Block 4)
100
101
110
111

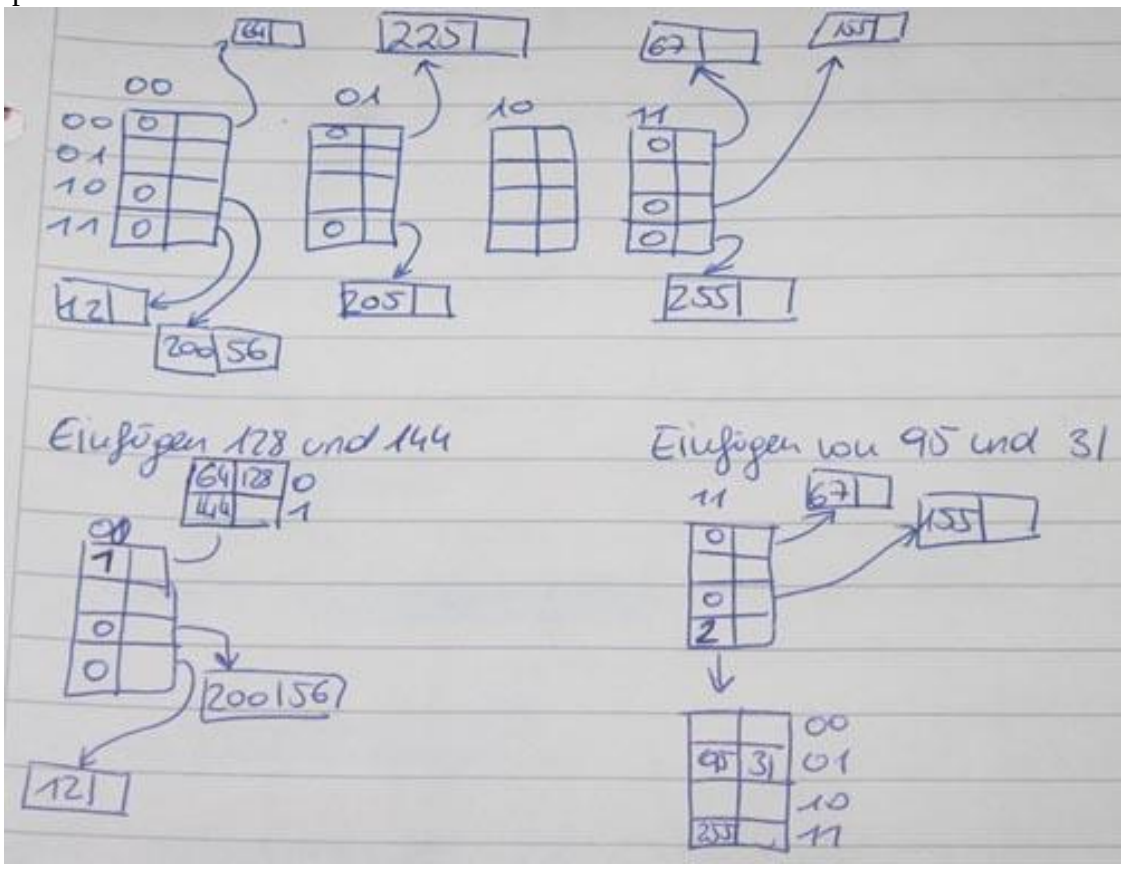
Lokale Tiefe (t) bei Block 1, 2, 3 und 4 jeweils 2
Jetzt wieder wie bei Fall 1 (t<d)

000 Link von 000 und 100 zu Block mit 8 (Block 1)
001 Link von 001 und 101 zu Block mit 9, 13 (Block 2)
010 Link von 010 und 110 zu Block mit 14, 6, 10, 26 (Block 3)
011 Link von 111 zu Block mit 7, 31, 23, 15 (Block 4)
100 Link von 011 zu Block mit 19 (Block 5)
101
110
111

Lokale Tiefe (t) bei Block 1, 2 und 3 jeweils 2 und bei 4 und 5 jeweils 3

Bounded Index Size E.H.

Beispiel:



Graphen (Kapitel 6)

Allgemein

- Knoten: v
- Kanten: e (z.b. e1 = [v1,v2] heißt das die Kante e1 von Knoten v1 zu v2 ist)
- Ungerichtet: keine Pfeile
- Gerichtet: Pfeile zeigen an in welche Richtung es zeigt
- Knotenzug: ist eine Kantenfolge, in der alle Kanten verschieden sind.
- Ein Weg oder Pfad ist eine Kantenfolge in der alle Knoten verschieden sind
- Ein Baum ist ein verbundener kreisloser Graph
- Alle Knoten die durch einen Weg verbunden werden können, heißen Komponenten eines Graphen

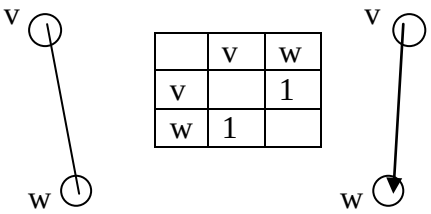
2 Methoden zur Speicherung:

- Adjazenzmatrix
- Adjazenzliste

Adjazenzmatrix

- Knoten repräsentieren Indexwerte einer 2- dimensionalen Matrix
- Kleine Graphen mit vielen Kanten
- Überprüfung von Adjazenzeigenschaft 0(1)
- Manche Algorithmen einfacher
- dfs schlecht, Rechenaufwand 0 ($|V|^2$)
- quadratischer Speicheraufwand 0 ($|V|^2$)

Bsp mit A[v,w] und A[w,v]:



	v	w
v		1
w	1	

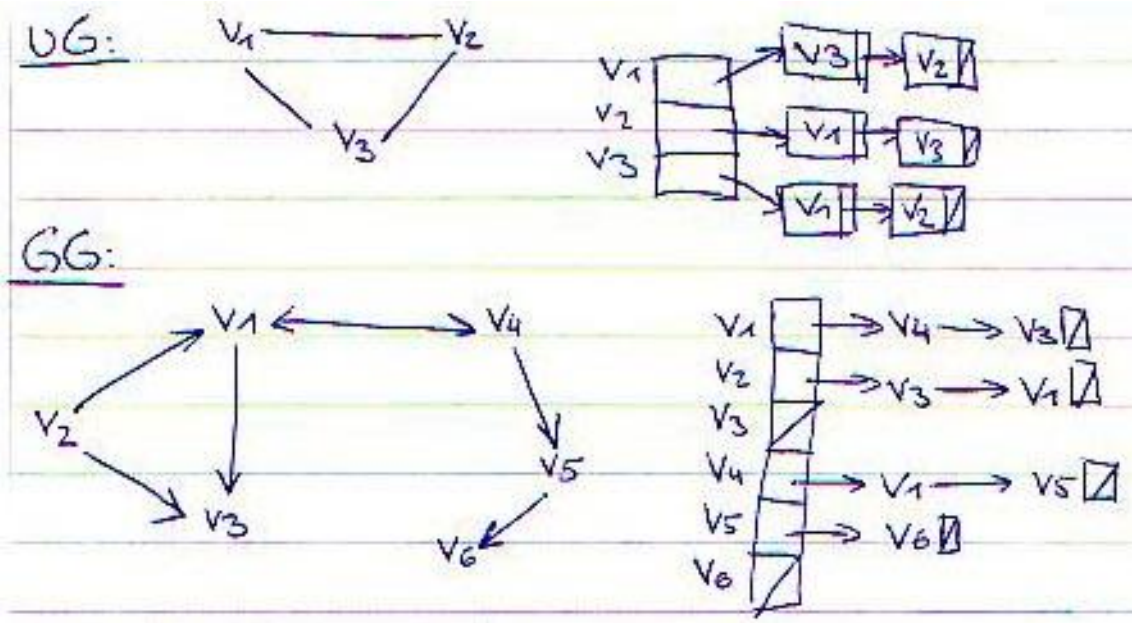
Bsp mit A[v_g,w]

	v	w
v		1
w		

Bei der 2ten Tabelle ist die Stelle v/w leer da es gerichtet ist!

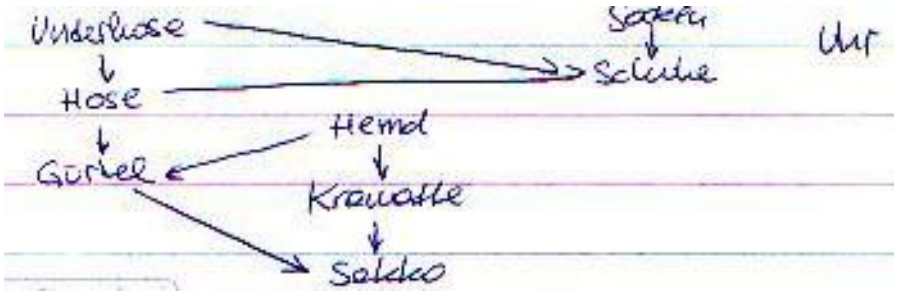
Adjazenzliste

- Für jeden Knoten werden alle adjazenten Knoten in einer Liste gespeichert.
- Gut für Graphen (G).mit wenigen Kanten
- Linearer Speicheraufwand 0(n)
- Tiefensuche (dfs) gut, Rechenaufwand 0 ($|V| + |E|$)
- Überprüfung Adjazenzeigenschaften 0 ($|V|$)
- Manche Algorithmen komplexer

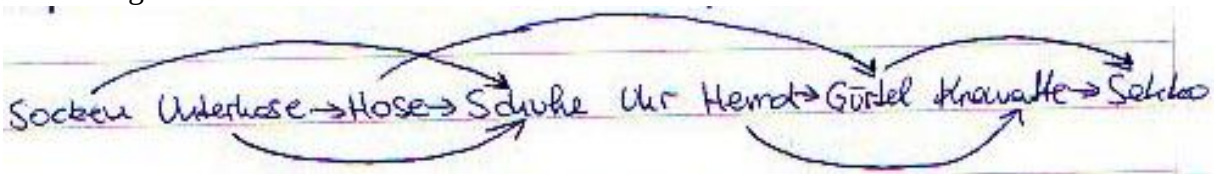


Topologisches Sortieren

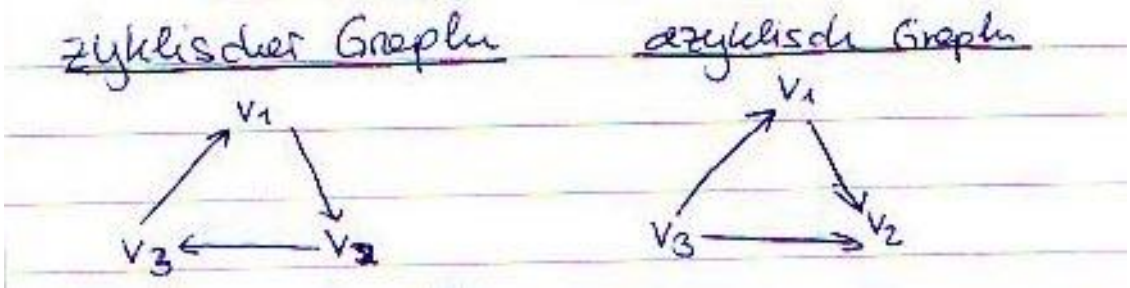
- Man muss beachten ob es eine Reihenfolge gibt



Eine Topologische Ordnung eines gerichteten azyklischen Graphen G ist eine lineare Anordnung aller Knoten.



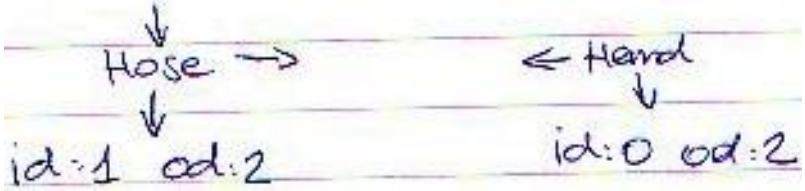
Wenn der Graph nicht azyklisch ist gibt es keine topologische Ordnung!



Mögliche Vorgehensweiße:

1. Suche Knoten aus den nur Kanten herausgehen
2. Ordne ihn in der topologischen Ordnung an.
3. Lösche ihn aus Graph
4. Weiter bei 1

Indegree: Anzahl der in v einmündenden Kanten
Outdegree: Anzahl der von v ausgehenden Kanten

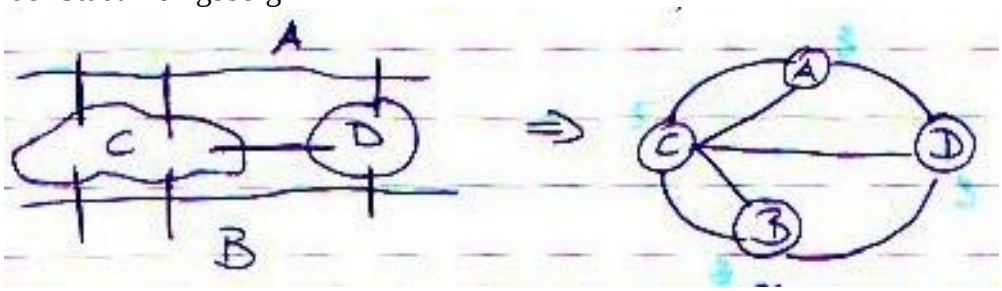


Traversieren eines Graphen

- Systematische und vollständige Besuchen aller Knoten des Graphen

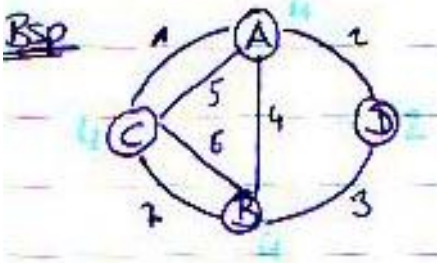
1. Tiefensuche depth- first- search (dfs)
2. Breitensuche Breadth- first search (bfs)

Problem der Stadt Königsberg



Gibt es einen Kreis im Graphen der alle Kanten genau einmal enthält?

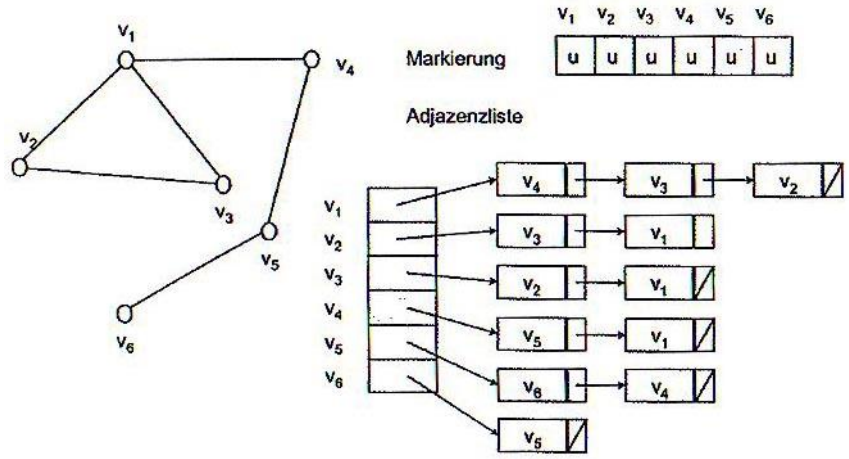
Euler: Ja, wenn alle Knoten gerade Knotengrade haben=> für Königsberg geht es nicht

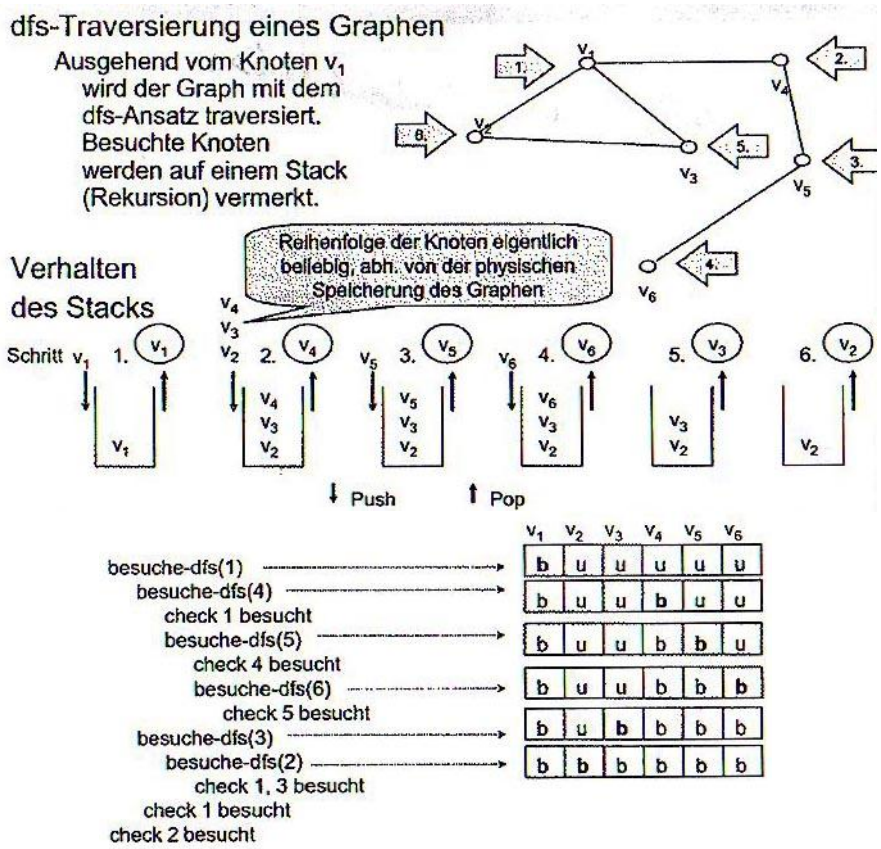


Depth first Search

- Wir interpretieren den Graphen als Labyrinth, wobei die Kanten Wege und die Knoten Kreuzungen darstellen
- Versuchen einen möglichst langen neuen Pfad zurückzulegen. Wenn wir keinen neuen Weg mehr finden, gehen wir zurück und versuchen den nächsten Pfad
- Wenn wir zu einer Kreuzung kommen, markieren wir sie (im Labyrinth durch einen Stein). Wir merken uns mögliche Pfadalternativen

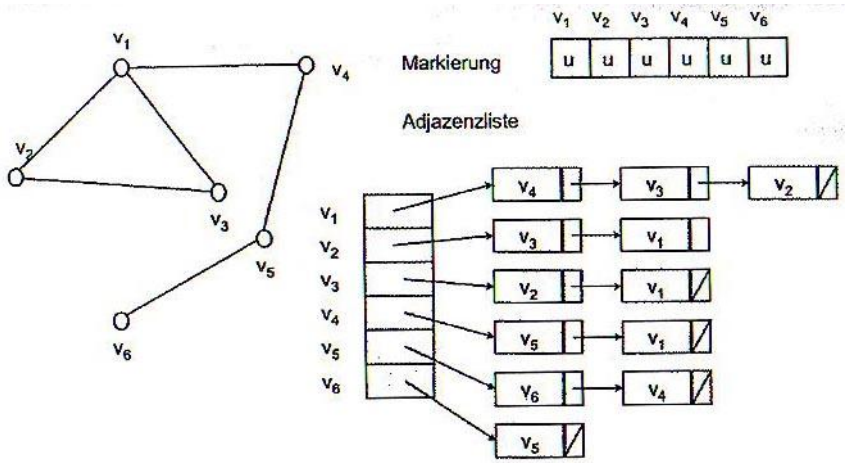
- Kommen wir zu eine markierten Kreuzung über einen noch nicht beschrittenen Weg, gehen wir diesen Weg wieder zurück (wir waren schon mal da)

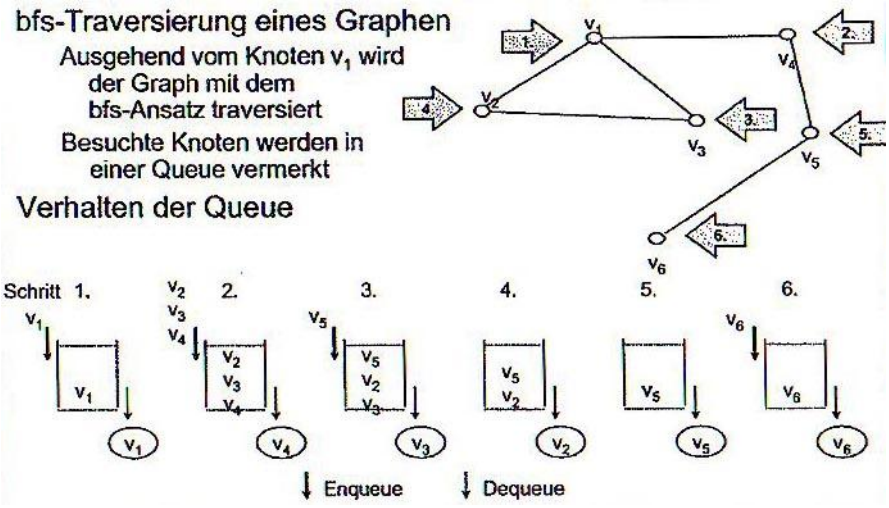


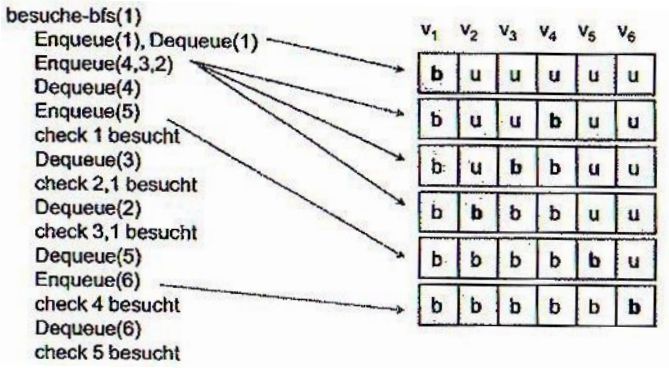


Breadth first Search

- Man leert einen Topf mit Tinte auf den Startknoten. Die Tinte ergießt sich in alle Richtungen (über alle Kanten) auf einmal
- Es werden alle möglichen Alternativen auf einmal erforscht, über die gesamte Breite der Möglichkeiten
- Die bedeutet, dass zuerst alle Möglichen, von einem Knoten weggehende Kanten untersucht werden, und danach erst zum nächsten Knoten weitergegangen wird







Aufwand von dfs und bfs

- Aufwandabschätzung abhängig von der Speicherungsform des Graphen. Bestimmend Aufwand für das Finden der adjazenten Knoten
- Adjazenzliste: dfs und bfs: 0 ($|V| + |E|$). Jeder Knoten wir einmal besucht, Adjazenzlisten werden iterativ abgearbeitet. Bei vollständig verbundenen Graphen 0 ($|V^2|$) da jeder Knoten $|V|$-1 Kanten besitzt daher $|E|=|V|^2$
- Adjazenzmatrix: dfs und bfs: 0 ($|V^2|$). Aufwand zum Finden des adj. Nachfolgereines Knoten 0 ($|V|$)

!!! Ab hier noch zu machen (Seite 443 und folgend)!!!

Weiterführende Links + Informationen

Zusammenfassung: Martin Tintel
Neuste Version: http://www.informatik-forum.at/showthread.php?t=48143

Seiten die im Skript fehlen: 237 bis 244 und 277 bis 284.

Bei den ausgearbeiteten Beispielen heißt die Farbe Rot, dass es sich hierbei um ein ungeklärtes Problem handelt. Grün bedeutet, dass eine Fixposition gefunden wurde, und man somit an der Stelle nichts mehr tun muss. Orange bedeutet, dass es hier zu einem Konflikt/Problem kommt oder eine Überprüfung stattfindet (und sich was ändern wird) und man deswegen dann z.b. einen Baum neu sortieren muss, Elemente austauschen muss, 2 Werte überprüfen muss,… damit kein Problem mehr auftaucht oder man z.b. eine Fixposition findet (und somit dann getauscht und grün markiert wird).

Übersicht bei Wikipedia:
http://de.wikipedia.org/wiki/Kategorie:Datenstruktur
http://de.wikipedia.org/wiki/Kategorie:Suchbaum
http://de.wikipedia.org/wiki/Kategorie:Datenbank
http://de.wikipedia.org/wiki/Sortierverfahren

Gute Webseiten zum Thema:
http://www.inf.fh-flensburg.de/lang/algorithmen/algo.htm
http://www.linux-related.de/index.html?/coding/coding_main.htm
http://lcm.csa.iisc.ernet.in/dsa/dsa.html

Beispiele/ Lösungen/ Applets/ …. die man brauchen könnte:
http://www.informatik.uni-leipzig.de/~wittig/Seminarmaterialien/AuD1_0405/
http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm
http://alfi.ira.uka.de/lehre/sommer2002/AVLTreeApplet/avl.html

Alte Unterlagen vom Schikuta:
http://leonardo.pri.univie.ac.at/~schiki/unterlagen/GAlgoDat/ und
http://www.pri.univie.ac.at/~schiki/unterlagen/

Weitere praktische Infos:
http://de.wikipedia.org/wiki/Landau-Symbol
http://de.wikipedia.org/wiki/Worst-Case-Laufzeit

TEMP/ TODO:

- vielleicht fehlt noch was aus Vektorkapitel (5)
- Laufzeitanalyse
- **liesen (01:41 PM) :**
- logb a = x
- b hoch x = a
-
-
- **liesen (01:41 PM) :**
- dadurch kann man x ausrechnen. das vergleicht man mit dem c (alles master theorem sachen) und dann hat man einen der drei fälle.

- **liesen (01:41 PM) :**
- das is nur wenn die rekkurenzgleichungen gegeben sind.
- **Da Lord (01:41 PM) :**
- **aso...**
- **Da Lord (01:41 PM) :**
- **hoffe echt das kommt net...**
- **Da Lord (01:42 PM) :**
- **aber wenn jeder sich was wünscht was nicht kommt und das auch so gemacht wird, dann kann gar nichts mehr kommen...**
- **liesen (01:44 PM) :**
- ich find das is totaaal einfach.
- **liesen (01:46 PM) :**
- einfach in die master theorem formel einsetzen.
- zb
- laut angabe: algorithmus A´: $T(n) = 9*T(n/3)+n^2$

- die mastertheoremformel sagt dann a = 9, b=3, c=2

- das setzt du in den logarithmus ein. log zur basis b von a = x

- dann hat man: log zur basis 3 von 9 = x

- 3^x = 9.. also ist x = 2, weil $3^2 = 9$

- dann vergleicht man das c und das x.

- c war 2 = 2 -> fall 2: t(n) = n^c *log n