Sortierverfahren

Quick-Sort	-gutes Laufzeitverhalten $n \cdot \log(n)$ -rekursive (absteigende) Struktur -'median' Suchen – ungefähr mittlerer Wert -alle Elemente die kleiner sind als Median werden links von Median einsortiert -alle Elemente die größer sind werden rechts von Median einsortiert -dadurch Aufteilung in Untergruppen -dann erneut 'median' suchen und sortieren -Abbruch wenn nur noch Vektoren mit einem oder keinem Element vorhanden sind → keine weitere Sortierung notwendig
Insertion Sort	-Aufwand : n^2 -an intuitives Verhalten von Menschen bei Sortierung angelehnt -noch nicht sortiertes Element wird genommen -Überprüfung ob linker Nachbar des Elements größer ist: 1.) wenn ja, dann linker Nachbar nach rechts schieben und Element mit neuem linken Nachbarn vergleichen 2.) wenn nein, Element stehen lassen und nächstes unsortiertes Element auswählen
Linear Sort (linsort)	Aufwand linear -Bsp. Postleitzahlen -es wird mit letzter Ziffer begonnen → Postleitzahlen werden in Stapel abhängig von ihrer letzten Ziffer unterteilt -anschließend Stapel aufsteigend untereinander zurückgestapelt und mit nächster Ziffer fortfahren -es müssen zu keiner Zeit zwei PLZ miteinander verglichen werden -es werden n Durchläufe benötigt, in jedem Durchlauf wird jede PLZ einmal verarbeitet -Problem: es wird zusätzlicher Speicherplatz benötigt u. ist nicht immer anwendbar
Top-Sort	-Verwendung bei flexiblen, dynamischen Datenstrukturen -gewisse Paare sind vergleichbar aber nicht alle Sortierung nach Relationen → 3 Axiome Bsp: **x,y,z** → Werte der Menge **S** 1.) Transitivität → **x<y** und **y<z** dann **x<z** 2.) Asymmetrie → wenn **x<y** dann <u>nicht</u> **y<x** 3.) Irreflexivität → <u>nicht</u> **x<x**

Anmerkungen:
- linear sort schneller, warum nicht diesen?
- grund: zusätzlicher speicherbedarf, schlüssel
- quicksort vorteil? schnell

Laufzeitkomplexität

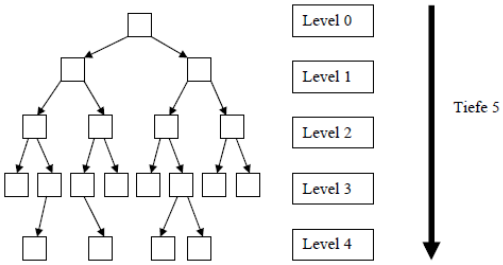
exponentiell	-abgestuft nach Größe der Basis -inakzeptabel wachsender Zeitbedarf -vermeiden wo immer es möglich ist
polynominal	-abgestuft nach höchster vorkommender Potenz -akzeptabel wachsender Zeitbedarf -höchste Potenz so niedrig wie möglich halten
logarithmisch	-gleichwertig, unabhängig von der Basis -sehr moderates Wachstum -wünschenswert
konstant	-beste Laufzeiteigenschaften -kommt in der Regel aber nicht vor

Bäume

Sortierte Bäume

Aufsteigend Sortiert:
Für jeden Knoten **K** gilt, **X<K** für alle Knoten **X** im **linken** Teilbaum des Knoten **K** und **X>K** für alle Knoten **X** im **rechten** Teilbaum des Knoten **K**

Absteigend Sortiert:
Für jeden Knoten **K** gilt, **X<K** für alle Knoten **X** im **rechten** Teilbaum des Knoten **K** und **X>K** für alle Knoten **X** im **linken** Teilbaum des Knoten **K**



Traversierung von Bäumen

wie traversiert man bäume?
-pre, in, post
-action an drei versch. stellen
-levelorder (stack...)
-rekursion

Traversierung → systematisches aufsuchen aller Knoten eines Baumes um bestimmte Operationen durchzuführen.

Preorder	-inder Hierarchie absteigend ein Knoten besuchen
	-Bearbeitung durchführen
	-nächste Knoten bearbeiten
	-zuerst linker Teilbaum, dann rechter Teilbaum unterhalb des bearbeiteten Knoten besuchen
	Code:
	`if( n)`
	`{`
	`action( n);`
	`preorder( n->left);`
	`preorder( n->right);`
	`}`

Inorder	-Bearbeitung eines Knotens zwischen der Bearbeitung seiner angehängten Teilbäume
	-Bearbeitung zwischen Traversierung von linkem u. rechtem Teilbaum
	Code:
	`if( n)`
	`{`
	`inorder( n->left);`
	`action( n);`
	`inorder( n->right);`
	`}`

Postorder	-zuerst angehängte Teilbäume eines Knotens traversieren bevor Knoten selbst bearbeitet wird
	-Bearbeitung nach Traversierung linker Teilbaum, rechter Teilbaum.
	Code:
	`if( n)`
	`{`
	`postorder( n->left);`
	`postorder( n->right);`
	`action( n);`
	`}`

Level Order	-Abarbeitung des Baums Ebene für Ebene -zuerst linke Seite, dann rechte Seite einer Ebene(Level) bearbeiten

Gemeinsamkeiten: zuerst linken, dann rechten Zweig besuchen
Unterschied: Position an der action()-Funktion ausgeführt wird

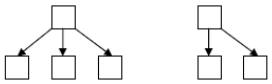
Anwendung: Huffmann-Codierung und Codekomprimierung, Binary Decision Trees

Unterschied Baum – Graph

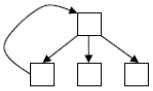
- baum keine schleifen - eindeutiger weg zum Blatt
- graph hat schleifen

Baum: Jeder Knoten in einem Baum lässt sich von der Wurzel aus auf genau einem Weg erreichen, daher sind folgende Kombinationen nicht möglich:

Kein Zusammenhang

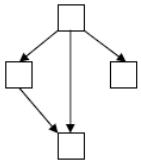


Zyklische Verkettung



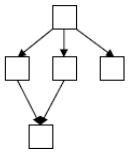
bei Graphen möglich

Überspringen von Knoten



bei Graphen möglich

Verschmelzen von Teilbäumen



bei Graphen möglich

Graphen

-Ein Graph der unzerlegbar und Schleifenfrei ist heißt **Baum**
-Ein Graph heißt **Wurzelbaum**, wenn es von einem Knoten w(Wurzel) zu jedem Knoten k des Graphen genau einen Weg gibt.

Gerichteter Graph	Eine Kante (a,b) verbindet den Knoten A mit Knoten B, aber nicht umgekehrt A → Startknoten B → Endknoten
Ungerichteter Graph (Symmetrischer Graph)	Eine Kante (a,b) verbindet sowohl den Knoten A mit dem Knoten B, als auch den Knoten B mit dem Knoten A

Wege in Graphen

Geschlossen, Schleife	Anfangs- u. Endknoten des Weges sind gleich
Schleifenfrei	Alle Knoten des Weges sind voneinander verschieden
Kantenzug	Alle zur Verbindung der Knoten verwendeten Kanten sind voneinander verschieden
Geschlossener Kantenzug	Kreis

Darstellung von Graphen in Datenstrukturen

Adjazenzmatrix	-Matrix mit Dimension = Anzahl Knoten -gibt es zwischen zwei Knoten eine Verbindung wird eine **1** eingetragen -gibt es zwischen zwei Knoten keine Verbindung wird eine **0** eingetragen -Nur rentabel wenn Kantenanzahl zu Knotenanzahl nicht zu gering -Speicherplatz: n^2
Adjazenzliste	-Zu jedem Knoten wird eine Liste gespeichert, die seine unmittelbaren Nachfolgerknoten enthält -Liste mit Verweis auf Liste (sehr dynamisch) -Vector mit verweis auf Liste (mittlerer Flexibilität) -2 Vektoren (geringe Flexibilität) -rentabel wenn Kantenanzahl zu Knotenanzahl gering -Speicherplatz: n+m
Inzidenzmatrix	-Matrix mit Knoten als Zeilen und Kanten als Spalten -ist ein Knoten ein Startpunkt einer Kante wird eine 1 eingetragen -ist ein Knoten ein Endpunkt einer Kante wird eine -1 eingetragen -erlaubt die Darstellung von Parallelkanten aber keine Schliefen -Speicherplatz: n*m

Kantentabelle	-Matrix(Tabelle) mit Kanten als Spalten und Start- und Endknoten als Zeile -Anfangs- und Endknoten jeder Kante wird in Tabelle festgehalten -eignet sich zur Darstellung von Graphen mit Parallelkanten u. Schlingen -Speicherplatz: 2*m
Wegematrix	-Matrix der Dimension = Anzahl Knoten - gibt es einen Weg von Knoten A nach Knoten B wird eine 1 eingetragen, sonst einen 0 -Wegematrix muss aus Adjazenzmatrix gebildet werden können -Multiplikation der Adjazenzmatrix mit sich selbst ergibt Wegematrix

Traversierung von Graphen

Traversierung von Graphen → führt zu einem aufspannenden Baum

Vorgehensweise:
Es werden Stück für Stück alle Knoten besucht und als besucht markiert. Kommt man wieder zum Ausgangsknoten oder ist ein Durchgang abgeschlossen und es wurden noch nicht alle Knoten besucht, so wird der nächste nicht besuchte Knoten gesucht und dort fortgefahren. Wenn alle Knoten einmal besucht wurden ist die Traversierung abgeschlossen.

Minimal aufspannende Bäume

-Spannbaum: von der Anzahl der Kanten her nicht verkleinerbarer zusammenhängender Teilgraph
-erreicht alle Knoten des ursprünglichen Graphen
Spannbäumen minimaler Länge:
-Anwendung: in vorhandenen Datennetz, Städte mit möglichst kurzen Kabeln zu verbinden
→ **Algorithmus von Kruskal**

Algorithmen zur Bestimmung des kürzesten Weges in Graphen

Aufgabe 1: Finde u. beschreibe den kürzesten Weg von einem Knoten A aus, zu allen anderen Knoten in einem Graph
Aufgabe 2: Finde und beschreibe die kürzesten Wege für alle Knotenpaare in einem Graphen

Floyd	-Aufwand: n^3 -löst Aufgabe 2 -knotenorientiert -für Navigation in Straßennetzen geeignet
Dijkstra	-Aufwand: n^2 -löst Aufgabe 1 -knotenorientiert -Datenstruktur: Adjazenzmatrix -Laufzeit unabhängig von der Anzahl der Kanten -gut geeignet für dicht besetzte Adjazenzmatrizen
Ford	-Aufwand n*m -löst Aufgabe 1 -kantenorientiert -Datenstruktur: Kantentabelle -Laufzeit abhängig von Anzahl der Kanten -gut geeignet bei wenig Kanten im Verhältnis zu Knoten

Tabelle Kap6_2.pdf Seite 141

Die folgende Tabelle stellt die verschiedenen Datenstrukturen und ihren Speicherplatzbedarf
für einen Graphen mit n Knoten und m Kanten
gegenüber:

Struktur	Speicher-komplexität	Vorrangige Verwendung	Einschränkungen
Adjazenzmatrix	n^2	Knotenorientierte Verarbeitung in Graphen mit relativ vielen Kanten im Verhältnis zur Knotenzahl	Keine Parallelkanten
Adjazenzliste	n+m	Knotenorientierte Verarbeitung in Graphen mit relativ wenig Kanten im Verhältnis zur Knotenzahl	
Inzidenzmatrix	n*m	Praktisch von geringer Bedeutung	Keine Schleifen
Kantentabelle	2*m	Kantenorientierte Verarbeitung in beliebigen Graphen	