ALGORITHMEN UND DATENSTRUKTUREN

Begriffsbildung

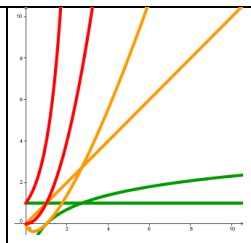
Algorithmus	Schrittweises präzises Verfahren zur Lösung eines Problems
Zeitkomplexität	sagt etwas über die Laufzeit
Speicherkomplexität	sagt etwas über den Speicherbedarf

Vorgehen und Komplexitätsklassen

Anzahl Ausführungen abhängig von n
$2 * proz() + 3n * proz() + 2n^2 * proz()$

↓

O-Notation (Teile ohne Bedeutung bei grossen n werden weggelassen)
$Schritte = 2\boldsymbol{n^2} + 3n + 2$
$\rightarrow O(n^2)$



exponentiell	$O(2^n)$
polynominal	$O(n^k)$
logarithmisch	$O(n * \log n)$
linear	$O(n)$
logarithmisch	$O(\log n)$
konstant	$O(1)$

Gleichwertige Lösungen

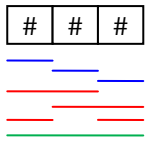
Maximale Teilsummen

Problem: Welche Teilfolge hat die grösste Summe

Tag	1	2	3	4	5	6	7	8	9	10
Gewinn	+5	-8	+3	+3	-5	+7	-2	-7	+3	+5

Lösungen

1. Intuitive Lösung
Jede Teilsumme wird berechnet.



Zeitkomplexität: $O(n^3)$
Speicherkompl.: $O(1)$

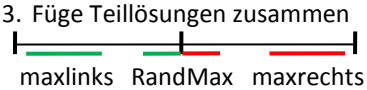
2. Zeit für Raum
Teilsummen zwischenspeichern

Länge Von\	1	2	3
1	#	#	#
2	#	#	
3	#		

Zeitkomplexität: $O(n^2)$
Speicherkompl.: $O(n^2)$

3. Teile und Herrsche
„divide et impera“ in Teilprobleme aufteilen

1. Teile dein Problem
2. Löse jedes Teilproblem
3. Füge Teillösungen zusammen



Zeitkomplexität: $O(n * \log n)$
Speicherkompl.: $O(n)$

4. Optimale Lösung
Jedes Element nur 1x auffassen

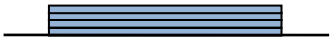
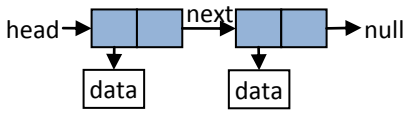
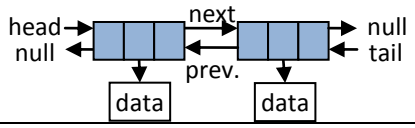
1. max. Teilsumme linker Teilfolge
2. rechtes Randmax. linker Teilfolge

Zeitkomplexität: $O(n)$
Speicherkompl.: $O(n)$

Definitionen

Randfolge	 Teilfolge, die irgendwo beginnt und am Rand aufhört.
Rechtes/Linkes Maximum	Maximale Summe aller möglichen Randfolgen.
Gleichwertigkeit von Algorithmen	Wenn sie für jede Eingabe dieselbe Ausgabe produzieren.
Gleichwertigkeit von Datenstrukturen	Wenn sie in jedem Fall die gleichen Resultate liefern.
Beweis -> Plausibilisieren durch Test	Allgemeiner Beweis zur Gleichwertigkeit ist nicht möglich.

Datenstrukturen („Behälterklassen“)

Reihung (=Array)	**Einfach verkettete Listen**	**Doppelt verkettete Listen**
Wie ein Papierstapel Zugriff nur auf oberstes Element	Variablen sind nur Referenzen auf Objekte (Ausnahme: Basisdatentypen) dynamisch (wachsen, schrumpfen); sind rekursiv	
 **Nachteil** Grösse festlegen, **Vorteil** Schneller Zugriff		
Einfügen: `put()` **Löschen:** `get()`	**Einfügen:** `insert()` `//vorne` `append()` `//hinten` **Löschen:** `delete()`	

Rekursion („Selbstaufruf“)

Lösungsmethoden

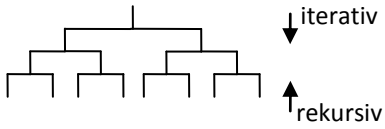
iterativ	schrittweise nacheinander
rekursiv	ineinander geschachtelt

Rekursiv definierte Folgen

Rekursionsbasis (Anfangsbedingung)	$a_0 = \#$
Rekursionsvorschrift (Folgeglieder)	$a_n = f(a_{n-1})$

Prinzip der Rekursion

Unterteile in Teilprobleme
Löse Baum von unten



Nachteile erhöhter Speicherbedarf
Permutationen {A, B, C} = ABC, ACB, BAC, BCA, …

Grundstruktur (im Pseudocode)

```
Programm Löse(Problem)
BEGIN
   IF(Problem einfach) THEN
      Löse //Rekursionsbasis
   ELSE
      Unterteile //Rekursionsvorschrift
      Löse(Teilproblem)
   END
END
```

Backtracking (try and error / Versuch und Irrtum)

Idee	Eigenschaften	Beispiele
• vor und zurückgehen bis Lösung gefunden • nach Versuch und Irrtum	• oft Rekursiv	• Labyrinth • Springerproblem • Damenproblem

Suchen

Internes Suchen	Alle Daten passen in den Speicher
Externes Suchen	Auslagerung auf Datenbanken

In Arrays und verketteten Listen

		Array (=Reihung)			Listen	
		unsortiert	**sortiert**		**unsortiert**	**sortiert**
		linear	linear	binär (teile und herrsche)	linear	linear
Suchen	Zeitkomplexität	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
	Speicherkompl.	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Einfügen	Zeitkomplexität	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$

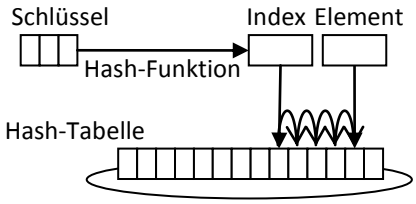
Hashtable

Ausgangslage: Objekte werden in einer Tabelle anhand von Schlüsseln (key, K) abgelegt.

Problem: Welche Datenstruktur??
Array -> Grösse unbekannt
Verkettete Liste -> langsam

Lösung: Hashtable
1. Einfügen: Aus Schlüssel wir ein Index berechnet
2. Solange an nächste Stelle gehen bis jene frei ist.

Idee: Mischung von Array + verketteter Liste



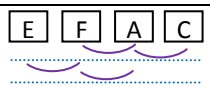
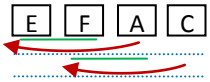
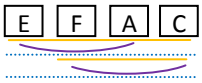
Sortierverfahren

Einführung

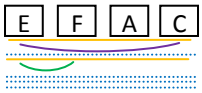
Definitionen

Internes Verfahren	Verfahren setzt voraus, dass alle Elemente im Speicher stehen, und die Operationen vergleichen und vertauschen existieren.
Externes Verfahren	Objekte auf externen Speicher, vertauschen gibt es nicht.
Stabililität	Ein stabiles Verfahren verändert die ursprüngliche Reihenfolge gleicher Schlüssel nicht.
Sortieralg. in Java	**Interface**: java.lang.Comparable; **Verwendung**: int compareTo(Object o) {...} **Sortieralgorithmus:** „tuned quicksort“
verwendete Operationen	vergleichen (häufiger, „billiger“) vertauschen

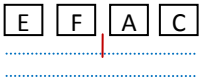
Quadratische Verfahren

		Zeitkomplexität			
		worst	**average**	**best**	**stabil?**
Bubblesort (Sortieren durch Vertauschen) Vertauschen benachbarter Elemente		$O(n^2)$	$O(n^2)$	$O(n)$	ja
Insertionsort (Sortieren durch Einfügen) Jedes Element an die richtige Stelle verschieben		$O(n^2)$	$O(n^2)$	$O(n)$	ja
Selectionsort (Sortieren durch Auswählen) Vertauscht im unsortierten Bereich das kleinste Elemente mit dem Ersten des unsort. Bereichs		$O(n^2)$	$O(n^2)$	$O(n^2)$	nein

Unterquadratische Verfahren

		Zeitkomplexität
Shellsort (Ähnlich dem Insertionsort) Sortieren mit abnehmender Schrittweite (7,3,1) Im letzten Schritt wandern die Elemente nicht mehr weit.	3: 1: 	schwierig: $O(n^{1+\varepsilon}), \quad \varepsilon > 0$

Rekursive Verfahren

		Zeitkomplexität			Speicherkomplexität			stabil?
		worst	**average**	**best**	**worst**	**average**	**best**	
Quicksort (teile und herrsche) Wähle element, Daten aufteilen, Teile sortieren		$O(n^2)$	$O(n * \log n)$		$O(n)$	$O(\log n)$		nein
Mergesort Teile in der Mitte, sortiere jede Hälfte füge Hälften zusammen (merge)		$O(n * \log n)$			$O(n^2)$	$O(n * \log n)$		ja

Logarithmische Verfahren

		Zeitkomplexität	stabil?
Heapsort (Haufen)	unsortierte Reihung → Halde (senken) → sortierte Reihung	$O(n * \log n)$	nein

Externe Verfahren

Problem: Daten passen nicht in Hauptspeicher

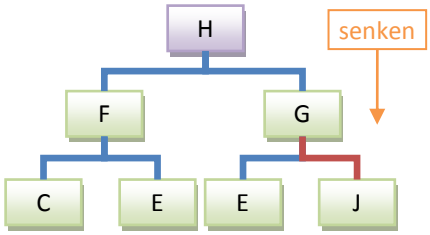
Lösung: Daten aufteilen, Teile im Speicher sortieren, anschliessend mischen (kleinstes herausnehmen)

	Zeitkomplexität	Speicherkomplexität	stabil?
Radixsort (z.B. PLZ) (jede Ziffer einzeln sortieren, beginnend bei der letzten)	$O(n * k)$ k = Anzahl Stellen	$O(n * \log n)$	

Bäume

Halde (heap)

Ansicht als Binärbaum



Ansicht als Reihung

H	F	G	C	E	E	J

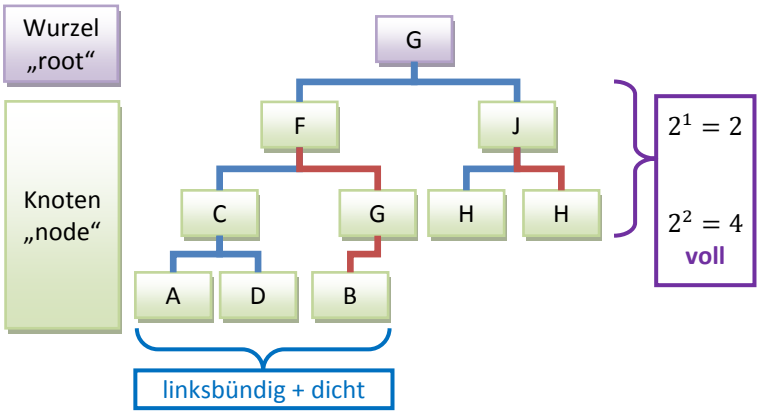
Vorgänger: $reihung[i/2]$
Nachfolger: $reihung[2i], [2i+1]$

Haldenbedingung:
Keine Komponente grösser als der Vorgänger

senken: Vorgang, der eine Halde, die an der Spitze „gestört“ ist (Fast-Halde), zu einer richtigen Halde macht

Haldenbedingung-Verletzung

Bäume



Binärbaum:
Jeder Knoten besitz max. 2 Nachfolger
Jeder Knoten hat genau einen Vorgänger (ausgenommen Wurzel)

voll:	wenn jede obere Ebene k genau 2^k Knoten besitzt
komplett:	wenn er voll ist und in der untersten Ebene alle Knoten „linksbündig + dicht“ sind
sortiert:	kein Knoten links hat einen grösseren Schlüssel kein Knoten rechts hat einen kleineren Schlüssel
streng sortiert:	gleiche Knoten müssen rechts sein
unsortiert	
Tiefe	= Anzahl Ebenen
Gewicht	= Anzahl Knoten

Übersicht

Suchen in sortierten Bäumen		
worst	average	best
$O(n)$	$O(\log n)$	

average	Suchen	Einfügen
sortiertes Array	$O(\log n)$	$O(n)$
sortierte verkettete liste	$O(n)$	$O(1)$
streng sortierter Binärbaum	$O(\log n)$	$O(1)$

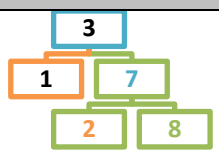
Sortieren mit Bäumen

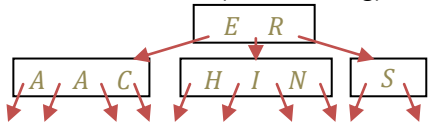
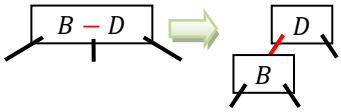
Darstellungsmöglichkeiten

mit Referenzen	**als Halde (heap)**
`class Node{ Node left, right; Comparable data;}`	
Vorteil: Einfach zum Vorstellen	Vorteil: kein Speicher für Referenzen

Baumsort

Durchwanderung		
inorder	links, operation, rechts	1 3 2,7,8
preorder	operation, links, rechts	3 1 7,2,8
postorder	links, rechts, operation	1 2,8,7 3



2-3-4 Bäume	**Rot-Schwarz-Bäume**	**B-Bäume**
Knoten können bis zu 4 Referenzen + 3 Schlüssel haben. (2ter Ordnung) 	**Idee**: 2-3-4 Bäume als Binärbaum  **Rot: Spezialkanten** **Schwarz: Originalkanten**	beliebig viele Schlüssel n-ter Ordnung hat m Schlüssel: $n \leq m \leq 2n$
Gutes Beibehalten der Ausgeglichenheit komplexe Implementation	längster Ast ist maximal doppelt so lang wie der kürzeste	