Grundlagen / Allgemein

Laufzeiten – Landausymbole

1, log n, n, n log n, n^2, n^3, 2^n

O Obere Grenze

$f(n) \in O(g(n))$

$\Leftrightarrow \exists c > 0, n0 \geq 1 \ \forall n \geq n0:$

$f(n) \leq cg(n)$

Für eine Konstante c wird f(n) ab einem n0 von g(n) dominiert

Ω Untere Grenze

$f(n) \in \Omega(g(n))$

$\Leftrightarrow \exists c > 0, n0 \geq 1 \ \forall n \geq n0:$

$f(n) \geq cg(n)$

Für eine Konstante c dominiert f(n) ab einem n0 die Funktion g(n)

Θ: $f(n) \in \theta(g(n))$

$\Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$

Die Funktionen f(n) und g(n) sind „gleichmächtig“

$3n^3 - 2n^2 + 4 \in O(n^3)$

$\exists c > 0, \exists n0 > 0 \ \forall n \geq n0:$

$3n^3 - 2n^2 + 4 \leq c \cdot n^3$

$3n^3 - 2n^2 + 4 \leq 3 \cdot n^3$

$4 \leq 2n^2$

n0 = 2 , c = 3

Komposition: Für Laufzeitfkt. f(n),g(n) einem Komplexitätssymbol M∈O,Ω,θ und einen Operator ∘ ∈ {+, *} def.:

$f(n) \circ M(g(n)) := M(f(n) \circ g(n))$

$M(f(n)) \circ M(g(n)) := M(f(n) \circ g(n))$

Programmiertechniken

Divide & Conquer:
Funktion wird meist rekursiv selbst aufgerufen
Problemgröße wird drastisch verkleinert
Abbruchbedingung sehr simpel

Memoization:
beschleunigt die Berechnung für referenziell unabhängige Funktionen, die öfter mit denselben Eingaben aufgerufen werden (speichert Werte zwischen)

Dynamische Programmierung
Fokus auf Kostenoptimierung
Voraussetzung:

- Teillösungen ergeben Gesamtlösung
- Teilprobleme überlappen

Greedy Methode

Teilschritte → optimale Teillösung
82€ gesucht mit 50,20,2,1€
Greedy: 50€,20€,6x2€ →8
Optimal: 4x20€, 2€ → 5

Speicherkomplexität

Frage: Aufrufe "verbrauchen" Speicher?
Operat. mit konst. Speicheraufwand O(1)
- Variablen deklarieren, Werte zuweisen
- Vergleiche und Grundoperationen
– For-Loops (wenn deren Inhalt konstanten Speicher benötigt).

Algorithmus benötigt konstant viel Speicher
→ arbeitet **In-place / in-situ**

Operationen die Speicher "verbrauchen" O(n), O(log n)
- Arrays vergrößern. Je nach Größe.
- Rekursionen (Linear zur Rekursionstiefe).

Wichtige Mathematische Ansätze

$$\sum_{k=a}^{b} 1 = b - a + 1 \qquad \sum_{k=a}^{b} k = \frac{(b-a+1)*(b+a)}{2} \qquad \sum_{k=1}^{n} k = \frac{(n \cdot (n-1))}{2}$$

Algorithmen

Suchwahrscheinlichkeit

→ mittlere Zugriffszahl:

m(x) = ∑(x∈S) p(x)*(id(x)+1) + ∑(x!∈S) p(x) *n

m Erfolgreiche Suche:

m(x|x ∈ S) = ∑ p(x) *(id(x) + 1) * (1/∑p(x))

m Nicht Erfolgreiche Suche

(x|x !∈ S) = n

→ „wie viele Felder bis leeres Feld“

Adaptive Verfahren: move to front, transpose (tauschen mit Vorgänger)

Hashing

1. expliziertes verketten: Jeder Tabellenplatz enthält eine Liste von Elementen; #Schlüssel →#Tabellenplätze (WorstC.O(n), BestC.O(1), AC.O(1+a) (a = Füllgrad = n/m)

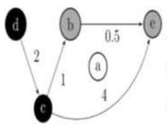
2. Offene Adressierung wenn #Elemente < #Tabellenplätze

→ linears Sondieren: Problem: Cluster/ Ketten, die Einfüge-/Suchzeit verschlechtern

→ double Hashing: Tabellenlänge Primzahl, zwei unanhängige Hash-Fkt. (Teilerfreme Restklassen → h1 % M , h2 % M-1)

Dijkstra Algorithmus

→ kürzester Weg im Graphen m.H.v. Queue
- keine Richtung berücksichtigt
- alle Richtungen gleich
- unnötige Knoten auch



Nodeu	a	b	c	d	e
pred[u]	-	c	d	-	c
dist[u]	∞	3	2	0	6

Huffmann-Code

→ Symbole/Char mit hoher Häufigkeit bekommen eine kleine Codierung (a=1,e=01,x=00001)

Sortieralgorithmen

Laufzeiten für Vergleiche (C) / Writes(W) / Komplexititätsklassen

Algorithmus	C:BC	C:AC	C:WC	W:BC	W:AC	W:WC	L:BC	L:AC	L:WC
Insertionsort	n-1	$\frac{(n-1)\cdot(n-2)}{2}$	$\frac{n^2-3n+2}{4}$	n-1	$\frac{n(n-1)}{2}$	$\frac{n^2+n-2}{4}$	O(n)	$O(n^2)$	$O(n^2)$
Selectionsort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$2(n-1)$	$2(n-1)$	$2(n-1)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubblesort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	0	$\approx \frac{3n(n-1)}{4}$	$\frac{3n(n-1)}{2}$	$O(n^2)$	$O(n^2)$	$O(n^2)$

Insertionsort: aktuelles Elem. in sortierten Bereich links einsortieren (stabil)

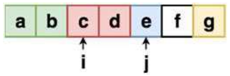
Selectionsort: select den nächst kleineren in hänge ihn rechts an den sortierten Bereich links (instabil)

Bubblesort: Mehrere Läufe über die Tabelle, benachbarte Elemente in falscher Reihenfolge tauschen (stabil)

Quicksort: Pivot-Element wählen, links alle kleineren, rechts alle größeren einsortieren → rekursiv anwenden (Instabil)

In-Place – Lomuto Partionierung:
Letzte Element ist Pivot Element
- i zeigt hinter Ende von linker Partition
- j zeigt hinter Ende von rechter Partition

L:BC	L:AC	L:WC
O(n*logn)	O(n*logn)	$O(n^2)$



Sortieralgorithmen Verkettete Liste

Mergesort: Teillisten(Länge 1) erstellen; zwei sortierte Teillisten werden zu einer sortierten Liste verschmolzen (stabil)

- Ist schneller als Quicksort, aber benötigt Zusatzspeicher

L:BC	L:AC	L:WC
O(n*logn)	O(n*logn)	O(n*logn)

Heapsort: Liste→ Heap→ delMax; (Instabil)
Max als Element ans Ende
Aufbauphase(n) + Auswahlphase(n*log n) = O(n*log n)

Countingsort(Strichliste): C=Countarray →1 ≤Ai≤k (stabil)
C Aufbauen O(n) + Ausgeben O(k)→ Zusammen: O(n+k)

Radix(-2)sort: Binäre Folge O(b*n) b-Bit; (stabil)
b konst. → kein zusätzlicher Speicher
sortiert nach MSB →2Teillisten MSB=0; =1
Teillisten r-sort nächst höchstwertigem Bit
Wenn nicht optimal b = log2n→ n*log n

Introsort: Quicksort, wenn entartete Teilliste→Heap (Instabil)

Timsort: Teilsequenzen Insert dann Merge

L:BC	L:AC	L:WC
O(n)	O(n*logn)	O(n*logn)

Stabiler Algorithmus: Reihenfolge von Daten mit gleichem Schlüssel bleibt gleich

ADT Liste	ArL	EV	DV
Speicherkomplex O(1)			
position first()	*O(1)*	*O(1)*	*O(1)*
position last()	*O(1)*	*O(1)*	*O(1)*
position next(elem)	*O(1)*	*O(1)*	*O(1)*
elem retrieve(posi)	*O(1)*	*O(1)*	*O(1)*
void delete(posi)	*O(n)*	*O(n)*	*O(1)*
void insert(posi, elem)	*O(n)*	*O(1)*	*O(1)*
void create()	*O(1)*	*O(1)*	*O(1)*
int length()			

ArL = Array List; EV = Einfach Verkettet, DV = Doppelt Verkettet

Datenstrukturen

ADT Stack (LIFO)
elem top()
elem pop() (& remove)
push(elem.)
bool empty()
Darstellung durch mind. eine Queue möglich

ADT Queue (FIFO)
elem front()
elem dequeue() (& remove)
enqueue(elem)
bool empty()
Darstellung durch mind. zwei Stacks möglich

ADT Set
void create()
value size()
bool member(elem)
remove(elem)
add(elem)

ADT Table	ArL	bS
void create()	O(1)	O(1)
value insert(key, elem)	O(1)	O(n)
remove(elem)	O(1)	O(n)
bool member(key)	O(n)	O(logn)

ArL = Array List; bS = Binäre Suche (Suchbaum)

Absatz zu Suchwahrscheinlichkeiten beachten(oben Mitte)

ADT PriorityQueue	ArL	Hea
void create()	*O(1)*	*O(1)*
insert(elem)	*O(n)*	*O(logn)*
elem delMin()	*O(1)*	*O(logn)*
elem delMax()	*O(1)*	*O(logn)*
bool empty()	*O(1)*	*O(1)*

ArL = Array List, sortiert, Hea = Heap

Graphen
- **(un)gerichtet**
- **Vollständig**

„zwischen jedem Knoten mind eine direkte Verbindung“
- **Zyklenfrei**
- **Valenz/Knotengrad**

- Ung.: #inzidenten Kanten
- Gerichtet: Eingangs-+Ausgangsgrad

Pfad
Folge von Knoten (Länge = #Kanten)
→ ***einfach Pf.***: Knoten nur 1x
→ ***Prim***: kürzeste Pfad
Zyklus
→ Pfad heißt Zyklus, falls Knoten Anfangs-und Endknoten eines Pfades
→ einfacher Zyklus, wenn einfacher Pfad → Kreis
Vollständiger jeder Knoten mit jedem anderen Verbunden
Clique → vollständiger Teilgraph
Stark Zusammenhängend / Starke Zusammenhangskomponente
Gerichtete Graphen: beide Richtung Kantenfolge zwischen Knoten oder Graph = 1Knoten
Breitendurchlauf (*BFS*) → Queue, Knotenfärbung(W,G,S)
Tiefendurchlauf (*DFS*) → *Stack* Knotenfärbung(W,G,S)
Wenn man einen Knoten im Stack trifft der grau gefärbt ist, ist der Graph zyklisch
Topologische Sortierung Reihenfolge, bei der alle vorhergehenden Bedingungen (Pfeile auf den Knoten) erfüllt sind, bevor er aufgeführt ist.

ADT Graph
.nodes() (Menge aller Knoten)
.edges() (Menge aller Kanten)
inAdjacent(knoten u) all eing. Knot.
outAdjacent(konten u) all ausg. Knot.

Adjazenz: Relation zwischen Knoten
Inzidenz: Knoten v ϵ Kante e
→ **Adjazenzmatrix & Adjazenzliste**

	A	B	C
A	0	0	0
B	1	0	0
C	1	1	0

A: ()
B: (A)
C: (A,B)

ADT Tree
create (elem root),
pos root(),
value retrieve (pos),
pos leftMost(poi),
pos nextLeft(pos),
pos nextRight(pos),
pos parent(pos),
void attach(pos, Teilbaum),
detach(pos) löscht Teilbaum,
bool empty()

Wald = Azyklischer ungerichteter Graph
Freie Bäume: zsm. hängender Wald
- Gewicht
- Verzweigungs Grad

Spannbaum → Teilgraph gleich # Knoten, kein Zylklus (Erzeuge Baum aus Graph)

Blatt: Knoten ohne direkten Nachfolger
Innerer/interner Knoten: Knoten, die keine Blätter sind
Äußerer/Externer Knoten = Leerer Knoten / NIL

Nachbarn, wenn gleicher Vorgänger und kein Teilbaum dazwischen
Höhe: Max(#Stufe) | binär: log2n ≤ h≤ n-1

Vollständiger binärer Baum, wenn jede Stufe maximal besetzt ist: d.h. Stufe i hat 2^i Knoten

Fastvollständiger binärer Alle bis auf letzte Stufe maximal besetzt ist

binärer Baum **saturiert**, wenn jeder Knoten entweder Blatt oder zwei nichtleere Teilbäume besitzt. Satuierte binäre Tree mit n >= 2 Blättern hat n-1 interne Knoten

Erweiterter binärer Baum = An jedem Blatt ein ext. Knot.

Catalanzahl: # der möglichen strukturell verschiedenen binär Bäume abhängig von der Knoten# n

$$B_n = \frac{1}{n+1}\binom{2n}{n}$$

Anzahl # möglicher geordneter Bäume bei Knoten

$$B_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

Bn > Bn-1

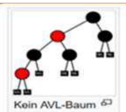
Suchbäume
→ Geordnet
Insert,member,delete(1,n)

Element Links vom Knoten ist kleiner Element rechts vom Knoten ist größer

Rot-Schwarz-Bäume
Binäre Suchbäume, bei denen alle Opertaionen auch im Worst-C. O(log n) sind. Es gilt:
1. Jeder Knoten ist rot oder Schwarz
2. jeder externe Knoten (null Zeiger) schwarz
3. wenn rot, dann ist direkter Nachfolger schw.
4. Alle Wege haben gl. Anzahl schw. Knoten

Insert: externen Knoten suchen & ersetzen(rot) + 2xNull-Zeiger(nur 3.Regel verletzt) 1.Wenn u Wurzel oder direkter Vorgänger von u ist schwarz → OK
2. Wenn direkter Vorgäng. v ist Wurzel → u schwarz
Sonst neu sortieren

!!!!!!!!!!!!
AVL=>R-S-Baum
R-S-Baum ≠> AVL

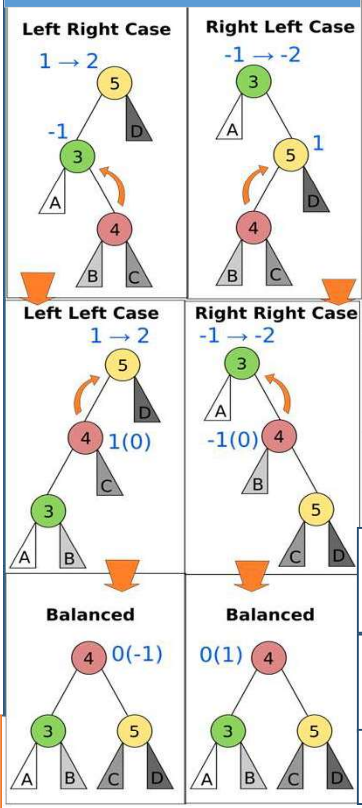


Baumdurchläufe:
Stufen Ordnung (Von O n. U. von L. n R.),
Präordnung (W,L,R),
Postordnung (L,R,W)
Symmetrische Ordnung / inorder (L,W,R)

Sequentielle Darstellung:
Sinnvoll bei statische Bäume, da platzsparende Speicherung.
Durchlauf + Strukturinformationen (Verzweigungsgrad, Gewicht, Klammerung d. Teilbäume, Blatt-, Nachbar-Bit(0,1) ->; ⊥)

AVL Bäume
- Ausgeglichener binärer Suchbaum
- an jedem internen Knoten v unterscheidet sich die Höhen des linken und rechten Teilbaums (Balancefaktor) um max 1

$$b(v) = |h(l) - h(r)|$$



Nach transformation von geordnetem Baum zu Binärbaum:
Präordnung -> Präordnung
Postordnung -> Sym. Ordnung

B Bäume (Mengen)
- vollständige Suchbäume
- Ordnung m>1
- Knoten hat 1 ≤ **#Schlüssel** ≤ m
- Interner Knoten mit ***n*** ***Schlüssel*** hat **n+1**(Zeiger) Nachfolger
- hoher Verzweigungsgrad
- #Schlüssel pro Knoten hoch
- Dateien alle in Blättern

B+ Bäume (Mengen)
B -Baum der Ordnung m > 1 ist ein B-Baum mit folgenden Eigenschaften:
- Die Wurzel 1 - 2m Schlüssel
- internen Knoten m - 2m Schlüssel
- Alle Wege von der Wurzel zu einem Blatt haben die gleiche Länge

Operationen: suchen/einfügen/ löschen →O(log(n)
Formeln:

Höhe h bei N Elementen:
$$1+\lfloor \log_{m+1}(N+1)\rfloor \ge h \ge \lfloor \log_{2m+1}(N+1)\rfloor$$

Best Case Perf./Max # Schlüssel
$$E = \sum_{k=0}^{h-1}(2m+1)^k \cdot 2m\,, h = Höhe$$

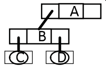
Max # an Knoten.
$$K = \sum_{k=0}^{h-1}(2m+1)^k\,, h = Höhe$$

Worst Case Perf./Min # Schlüssel
$$E = 1 + \sum_{k=0}^{h-2} 2\cdot m\cdot(m+1)^k\,, h = Höhe$$

Min # an Knoten.
$$K = 1 + \sum_{k=0}^{h-2} 2\cdot(m+1)^k\,, h = Höhe$$

Elemente / Blätter
#Blätter = #Schlüssel + 1

Verkettete Darstellung:
→ Zeigerdarstellung binär



→ gefädelte binäre Bäume

Arrayindex i bei (fast) vollständige binär Baum:
- Vorgänger: ((i+1):2) – 1
- Linker Nachfolger 2i + 1
- Rechter Nachfolger 2i +2

Heaps
- linksvollständiger binärer Baum
- an den Knoten sind Schlüssel angeheftet
- Wurzel ist Min / Max → Min-/ MaxHeap

Heap-Eigenschaft:
Schlüsselwerte auf dem Weg von der Wurzel zu einem Blatt sind
- monoton fallend (Max-Heap)
- monoton steigend (Min-Heap)

Einfügen → siftUp:
Element wird so lange mit seinem Vorgänger getauscht bis es an der richtigen Stelle im Heap ist
Wurzel löschen →siftDown:
letztes Element an die Wurzel ziehen und dann so lange mit den größten (kleinsten) Nachfolger tauschen bis es richtig ist (alles dadrunter muss monoton → O(log n)
→ **insert/delete**: O(log n)
→ **k-kleinstes Element** → k*Wurzel löschen → O(k*log n)

Heapeigensch. in Stufenord.
$X[\lfloor \frac{i+1}{2} \rfloor -1] \ge X[i], 1 \le i < N$ für Max

Heapeigenschaft in Array
a[i]≥ a[2i+1] & a[i]≥ a[2i+2] für

Für den Min-Heap müssen die Vorzeichen gedreht werden