Datenstrukturen

Laufzeiten – Landausymbole
1, log n, n, n log n, n^2, n^3, 2^n
O Obere Grenze
f(n) ∈ O(g(n))
⇔ *∃ c >0, n0 ≥ 1 ∀n≥ n0* :
f(n) ≤ cg(n)
Für eine Konstante c wird f(n) ab einem n0 von g(n) dominiert
Ω Untere Grenze
f(n) ∈ Ω (g(n))
⇔ *∃ c>0, n0 ≥ 1 ∀ n≥ n0*:
f(n) ≥ cg(n)
Für eine Konstante c dominiert f(n) ab einem n0 die Funktion g(n)
Θ: f(n) ∈ Θ(g(n))

⇔ *f(n) ∈ O(g(n)) ∧ f(n) ∈ Ω (g(n))*

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

$3n^3 - 2n^2 + 4 \in O(n^3)$
∃ c >0 , ∃ n0 > 0 ∀ n ≥ n0:
$3n^3 - 2n^2 + 4 \leq c*n^3$
$3n^3 - 2n^2 + 4 \leq 3*n^3$
$4 \leq 2n^2$
n0 = 2 , c = 3

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

ADT **verkettete** Liste (Laufzeit: O(x), Speicher: O(y))
position first() (1,1)
position last() (1,1)
position next(elem)
int length()
elem retrieve(posi) (1,1)
void delete(posi) (n,1)
void insert(posi, elem) (1,1)
//fügt elem hinter posi ein
void create() (1,1)

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

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

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

ADT Liste (Laufzeit: O(x), Speicher: O(y))
Array
position first() (1,1)
position last() (1,1)
position next(elem)
int length()
elem retrieve(posi) (1,1)
void delete(posi) (n,1)
void insert(posi, elem) (n,1)
//fügt elem hinter posi ein
void create() (1,1)

Sortieralgorithmen Verkettete Liste
Mergesort: Teillisten(Länge 1)erstellen;zwei sortierte Teillisten werden zu einer sortierten Liste verschmolzen(Laufzeit: n*log n), schneller als Quicksort, aber benötigt Zusatzspeicher
Heapsort: Liste→ Heap→ delMax;
Max als Element ans Ende
Aufbauphase(n) + Auswahlphase(n*log n)
= O(n*log n)

Sortieralgorithmen (Best-C.,Worst-C., Average-C.)
Insertionsort: aktuelles Elem. in sortierten Breich links (n,n*(n-1):2,n^2)
Selectionsort: select den nächst kleineren in hänge ihn rechts an den sortierten Bereich links(n^2,n*(n-1):2,n^2)
Bubblesort: tauscht den aktuellen Wert so lange durch, bis ein kleinere kommt und fängt dann von vorne an (n^2,n*(n-1):2,n^2)
Effektive: **Quicksort**: Pivot-Element egal wo-links alle kleineren, rechts alle größeren(nlogn,n^2,nlogn) Zusatzspeicher (Stack)
In-Place (unabhängig #Elemente)

Greedy Methode
Teilschritte → optimale Teillösung
82€ gesucht mit 50,20,2,1€
Greedy: 50€,20€,6x1€ →8
Optimal: 4x20€, 2€ → 5

Dijkstra Algorithmus
→ kürzester Weg im Graphen
- kleine Richtung berücksichtigt
- alle Richtungen gleich
- unnötige Knoten auch

Huffmann-Code
→ Symbole/Char mit hoher Häufigkeit bekommen eine kleine Codierung (a=01,e=10,x=01101)

Countingsort(Strichliste): C=Countarray →1 ≤Ai≤k
C Aufbauen O(n) + k-Schritte → O(n+k)
Radix(-2)sort: Binäre Folge O(b*n) b-Bit; 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 entartet→Heap
Timsort: Teilsequenzen Insert. dann Merge

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)

Algorithmen

Pfad
→ Folge von Knoten (Länge m = #Kanten)
→ *einfach*: Knoten nur 1x
→ ***Prim***: kürzeste Pfad
Zyklus
→ Pfad heißt Zyklus, falls Knoten a = K. e
→ einfacher Zyklus, wenn einfacher Pfad
Clique → nicht zsm.hängend
Vollständiger Teilgraph
Stark Zusammenhängend
Gerichtete Graphen: beide Richtung zwischen Knoten oder Graph = 1Knoten
Wald
Azyklisch ungerichteter Graph
Breitendurchlauf → *BFS*,
Knotenfärbung(W,G,S)
→ Realisierung mit Queue
Tiefendurchlauf → *DFS*
Dag → gerichtet&azykl.

Graphen
- **(un)gerichtet**
- **Vollständig**
„zwischen jedem Knoten mind eine direkte Verbindung"
- **Zyklenfrei**
- **Valenz/Knotengrad**
Ung.:#inzidenten Kanten
Gerichtet.Eingangs-+Ausgangsgrad

ADT Graph
.nodes() (Menge aller Knoten)
.edges() (Menge aller Kanten)
inAdjacent(knoten u) alle eingehenden Knoten zu u
outAdjacent(konten u)alle ausgehenden Knoten von u

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

$$\begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 0 & 0 \\ B & 1 & 0 & 0 \\ C & 1 & 1 & 0 \end{array}$$

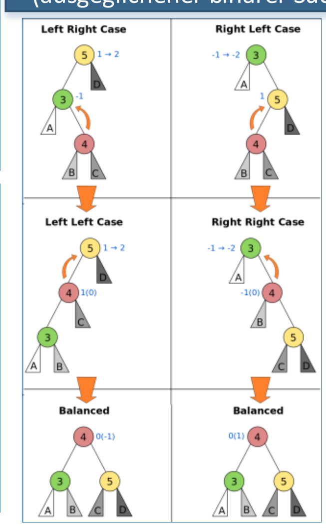
A: ()
B: (A)
C: (A,B)

Ein **Geordneter Baum** besteht aus einem Knoten (Wurzel/ root) r und n ≥ 0 Bäumen T_1 , … , T_n
(den Teilbäumen von r).
Wichtig für einen geordneten Baum:
i < j T_i links von T_j

Spannbaum
Ein spannender Baum(auch Gerüst) eines zusammenhängenden ungerichteten Grahen G ist ein Teilgraph mit gleicher Knotenmenge, der ein freier Baum ist.
„von r nur kürzeste Pfade zu jeden Knoten"

B+ Bäume(Mengen)
Ordnung m > 1 → vollständige Suchbäume: 1 ≤ **Schlüssel** ≤ m
Interner Knoten(HS) mit
N Schlüssel hat n+1(Zeiger) Nachfolger, → hoher Verzweigungsgrad, Schlüssel pro Knoten hoch
→ Dateien alle in Blättern
→ Root r: 1-2m Schlüssel
→ Interne Knoten: m - 2m
Alle Wege gleiche Länge
→ suchen/einfügen/löschen
→O(log(n))
$\lfloor \log_{2m+1}(N+1) \rfloor \leq h \leq 1 + \lfloor \log_{m+1}(N+1) \rfloor$

AVL Bäume
Ist ein binärer Suchbaum, bei dem sich an jedem internen Knoten v die Höhen des linken und rechten Teilbaums um max 1 unterscheiden (ausgeglichener binärer Suchbaum)



Größtes Element im linken Teilbaum rechts unten
Kleinstes Element im rechten Teilbaum links unten

Rot-Schwarz-Bäume
Sind 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

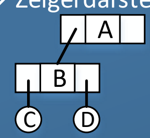
Insert: externen Knoten suchen &ersetzen(rot) + 2xNull-Zeiger(nur 3.Regel verletzt) 1.Wenn u Wurzel oder direkter Vorgänger von u ist schwarz → OK
2. Wenn direkter Vorgäng. v ist Wurzel → u schwarz
Sonst neu sortieren

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

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

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

Verkettete Darstellung:
→ Zeigerdarstellung binär



→ gefädelte binäre Bäume
Sequentielle Darstellung:
- statische Bäume
+ platzsparend, Durchlauf, Strukturinformationen (Verzweigungsgrad, Gewicht, Klammerung d. Teilbäume, Blatt-, Nachbar-Bit(0,1))
Baumdurchläufe:
→ Stufen(links vollständig)-, Prä-(W,L,R), Post(L,R,W)- und Symmetrische Ordnung (inorder|L,W,R)
→ Arrayindex i
→ (fast) vollständige binär
- Vorgänger((i+1):2) – 1
- Linker Nachfolger 2i + 1
- Rechter Nachfolger 2i +2

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()

Heaps
→ linksvollständiger binärer Baum
→ an den Knoten sind Schlüssel angeheftet
→ Wurzel ist Min / Max → Min-/ MaxHeap
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 fallend oder steigend sein) → O(log n)
→ **insert/delete**: O(log n)
→ **k-kleinstes Element** → k*Wurzel löschen → O(k*log n)

$X[\lceil \frac{i+1}{2} \rceil - 1] \geq X[i], 1 \leq i < N$

ADT PriorityQueue
(Array / sortiert)
void create(),
insert(elem) O(1)/n,
elem delMin()O(n)/1,
elem delMax(),
bool empty(),
Ein **Heap** ist eine Darstellung einer Pr.Queue

Heaps Array → a[i]≤a[2i+1] & a[i]≤a[2i+2]

Freie Bäume: zsm. hängender Wald
- Gewicht
- Verzweigungs Grad

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

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

Blatt: Knoten ohne direkten Nachfolger
Innerer Knoten: Knoten, die keine Blätter sind

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

Die Wurzel von Teilbäumen **direkte Nachfolger** von r
Alle Knoten der Teilbäume **Nachfolger** von r
Wenn v **direkter Nachfolger** von u, u **direkter Vorgänger** von v