

O-Notation

$g(n) = O(f(n)) \Leftrightarrow \left(g(n) \frac{1}{f(n)}\right)$ ist beschränkt (z.B. Konvergent)

Fester Wert $\Rightarrow g = O(f(n))$ UND $f = O(g(n))$;

Differenzengleichungen

1. Aus Angabe lesen:

b: Für Eingabe=1, $x_n = a_n x_{n-1} + b_n$ Einfache Sonderfälle:

2. $\pi_n = \prod_{i=2}^n a_i$

$$\bullet x_n = a_n x_{n-1} = b \prod_{i=2}^n a_i$$

3. $x_n = \pi_n \left(b + \sum_{i=2}^n \frac{b_i}{\pi_i} \right)$

$$\bullet x_n = x_{n-1} + b_n = b + \sum_{i=2}^n b_i$$

Nützliche Zahlen

$\sum_{i=1}^n \frac{1}{i} = H_n$ Abschätzung: $\ln(n+1) \leq H_n \leq \ln(n) + 1$

$$\sum_{i=1}^n H_i = (n+1)H_n - n$$

$x_n = \sum_{i=1}^{n-1} x_i + sth_n \Rightarrow x_{n-1} = \sum_{i=1}^{n-2} x_i + sth_{n-1} \Rightarrow \sum_{i=1}^{n-2} x_i = x_{n-1} - sth_{n-1}$
 $\Rightarrow x_n = 2x_{n-1} + sth_n - sth_{n-1}$

$\sum_{i=1}^n c = nc$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \sum_{i=1}^n i^2 = \frac{n(n+1)(n+2)}{6}$$

$$\sum_{i=1}^n (2i-1) = n^2 \quad \sum_{i=1}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$\sum_{i=1}^n \lfloor \log_2(i) \rfloor = (n+1) \lfloor \log_2(n) \rfloor - 2(2^{\lfloor \log_2(n) \rfloor} - 1)$

$$\bullet \sum_{i=0}^n c^i = \frac{c^{n+1}-1}{c-1}$$
$$\sum_{i=0}^n i c^{i-1} = \frac{(n+1)c^n(c-1) - (c^{n+1}-1)}{(c-1)^2}$$

rationale Summen

1. $\sum_{i=1}^n \frac{sth}{polynom}$

2. Partialbruchzerlegung: $\frac{x_i}{polynom} = \frac{a}{1.NST} + \frac{b}{2.NST} \dots$

Bei Mehrfachen NST ($k.NST$)^{Viel-fachheit} statt k. NST

3. Koeffizientenvergleich;

Logarithmen

$$\bullet \log(xy) = \log(x) + \log(y)$$

$$\bullet \log(x^c) = c \log(x)$$

$$\bullet e^{\log(x)} = x$$

Suche im Array

Sequentiell: gehe der reihe nach alle elemente durch bis das element gefunden $O(n)$

Binär: Sortiertes Array; betrachte mittleres Element;

<: Wiederhole im linken Teilarray, >: im rechten; worst-Case: $O(\log_2 n)$

Quick-Select: Suche k. kleinstes element: Analog zu probab. Quicksort, betrachte nur das Teilarray, in dem der Index k liegt; durchschnitt $O(n)$

Selection-Sort

Suche Minimum; Tausche es mit dem ersten Element; Wiederhole im Array $[2...n]$, $O\{n^2\}$

Bubble-Sort

durchlaufe elemente, tausche mit nachfolger wenn dieser größer; wiederhole $O\{n^2\}$

Quicksort

1. Wähle Pivot= letztes Element

2. lasse zeiger von beiden Enden des Restarrays nach innen laufen:

wenn der rechte zeiger auf ein kleineres bzw. der linke auf ein größeres Element als das Pivot zeigt stoppe den zeiger; wenn beide gestoppt: tausche sie, wenn sich die Zeiger treffen tausche das Pivot nach innen

3. Wiederhole Quicksort im Rechten und linken Teilarray.

zeit: best Case: $O(n \log_2 n)$, worst-Case $O(n^2)$

probabilistisch:: start: wähle zufälliges Element als Pivot und tausche ans ende

Heapsort

z.B. $[37, 45, 57, 59, 58, 99]$

Interpretation Array als binärer Heap;

$a[2i]$ und $a[2i+1]$ sind die Kinder von $a[i]$;

Array durchläuft die Ebenen von oben nach

unten, von links nach rechts

Heapbedingung: Vater \leq Kinder

Heapsort: Erzeuge Heap; Tausche letztes Element mit Wurzel, DownHeap in $[1...n-1]$, wiederhole bis array leer; worst-case: $O(n \log_2 n)$

Erzeuge Heap: Führe Einsichern für alle knoten durch (ebenenweise von unten rechts zur wurzel); $O(n)$

DownHeap: Knoten > Kind: Tausche mit Kleinerem nachfolger; wiederhole rekursiv

Revisited: Bestimme Pfad der kleineren Nachfolger bis zum Blatt, speichere den index.

\Rightarrow index des i. Knotens auf dem Pfad sind die vordersten i Bits des Blattindex

Lineare Suche:: Suche vom Pfadende aus die Einfügestelle, speichere Wurzel, alle Pfad-elemente rücken eine Ebene nach oben, einfügen Wurzel

binärsuche:: analog dazu, suche Einfügestelle mit binärer Suche im Pfad

Bäume

Preorder:: Knoten-linkerBaum-rechterBaum // Wurzeln sind links

Inorder:: linkerBaum-Knoten-rechterBaum

Postorder:: linkerBaum-rechterBaum-Knoten // Wurzeln sind rechts

Binärer Suchbaum

Knoten haben 2 Kinder, kleiner im Linken, größer im rechten Teilbaum

Suche: Starte bei Wurzel, wenn gesucht größer: rechts, wenn kleiner links, wiederhole bis gefunden oder Blatt

Einfügen: Suche im Baum; füge ein (achte auf links/rechts)

Löschen: Suche den Knoten; wenn:

- nicht vorhanden: nichts tun
- Blatt: Lösche Knoten, Referenz im Vorgänger auf null
- 1 Nachfolger: Referenz im Vorgänger auf nachfolger, lösche Knoten
- 2 Nachfolger: Tausche mit größtem Element im linken Teilbaum (symmetrischer Vorgänger), lösche Element

symmetrischer Vorgänger: einmal nach links, dann rechts solange möglich;

AVL-Baum

Bedingung: $|\text{Balancefaktor}| < 1$

Balancefaktor = $\text{Höhe}_{\text{rechterTeilbaum}} - \text{Höhe}_{\text{linkerTeilbaum}}$

$h < 1.45 \log_2(n+2) - 1.33 \Rightarrow$ höhe max. 45 % schlechter als best-case

Einfügen/Löschen: wie bei binär, anschließend Ausgleichen

Ausgleich nach einfügen: (Umgekehrt für +2)

Suche balancefaktor -2 am weitesten unten im Pfad zum eingefügten element

betrachte linken Nachfolger (b):

bei -1: Rechtsrotation um linken Nachfolger (a)

bei +1: Doppelrotation

Maximal einmal ausgleichen nötig

Ausgleich nach Löschen: (Umgekehrt für +2)

Suche balancefaktor -2 am weitesten unten im Pfad zum gelöschten element;

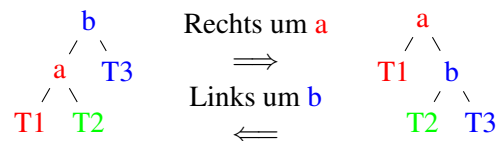
betrachte linken Nachfolger (a):

bei -1,0: Rechtsrotation um linken Nachfolger (a)

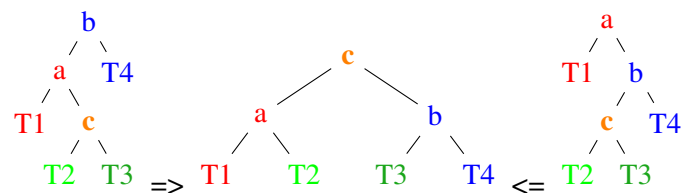
bei +1: Doppelrotation

wenn linker Nachfolger +1 oder -1 war, dann für höhere knoten vllt. weiter ausgleichen.

Rotationen



Doppelrotation



Treap

jedem element wird zusätzlich eine zufällige Priorität zugewiesen

Heapbedingung: $\text{Prio}_{\text{Vater}} < \text{Prio}_{\text{Kind}}$

\Rightarrow Treap ist eindeutig bestimmt

Erwartungswert Pfadlänge: $2 \frac{n+1}{n} H_n - 3$; Erwartungswert Rotationen: < 2

Einfügen: Analog Binärer Baum; danach: Rotation(nach oben) bis heapbedingung erfüllt

Löschen: Suche knoten, rotiere mit kleinerem Nachfolger (Priorität) bis Blatt; lösche **B-Bäume**

Entwickelt für Festplatten, minimieren zugriffe in datenbanken

Ordnung d \Rightarrow Knoten hat $\lceil \frac{d}{2} \rceil$ bis d Nachfolger, zwischen $\lfloor \frac{d-1}{2} \rfloor$ und d-1 Elemente,

Wurzel hat mind. 2 Nachfolger oder ist Blatt

alle Blätter sind immer auf einer Ebene \Rightarrow immer vollst. ausgeglichen

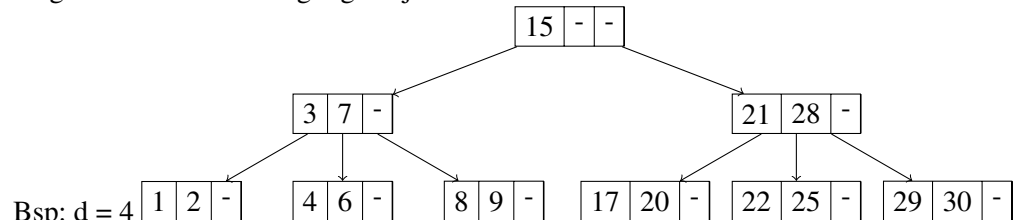
Baum mit höhe h hat mindestens $1 + 2 \frac{\lceil \frac{d}{2} \rceil^h - 1}{\lceil \frac{d}{2} \rceil - 1}$ und maximal $\frac{d^{h+1} - 1}{d - 1}$ Knoten

höhe ist $O(\log_2(n))$, genauer: zwischen $\log_d(n+1) - 1$ und $\log_{\lfloor (d-1)/2 \rfloor + 1} \left(\frac{n+1}{2} \right)$

Aufbau eines Knotens/Seite:

Adresse	Element	Adresse	...	Adresse
---------	---------	---------	-----	---------

Es gilt binärbaum Bedingung für jedes Element mit seiner rechte/linke Adresse



Einfügen: Suche; Wenn Blatt nicht voll: füge ein (sortierung beachten)

Wenn das Blatt voll ist: (ggf. rekursiv)

1. Suche das mittlere Element M_{itte} , die elemente rechts davon werden neues Blatt;
2. füge M_{itte} in den Vaterknoten ein, der rechte Verweis zeigt auf das neue Blatt;
3. füge das neue Element ein

Löschen:

1. Element nicht in einem Blatt \Rightarrow tausche es mit dem Nachfolger in Sortierreihenfolge (ist in einem Blatt); lösche;
2. ist Seite danach zu Klein ($< \lfloor \frac{d-1}{2} \rfloor$) versuche Ausgleich mit direkten Nachbarblatt: dazwischenliegendes Element (M_{itte}) kommt vom Vaterknoten in den zu kleinen Knoten, der Nachfolger/Vorgänger aus dem anderen in den Vaterknoten
3. Ausgleich nicht möglich: Füge 2 benachbarte Knoten + M_{itte} aus Vaterknoten zu einem Zusammen. Wiederhole ggf. rekursiv.

B*-Baum: Beim Einfügen in volle Seite versuche ausgleich mit direkten Nachbarn \Rightarrow bessere Speicherausnutzung

B+-Baum: Bei Datenbank-Indexen: nur die Suchschlüssel im B-Baum, Blätter zeigen auf die Datenblöcke, diese sind doppelt verkettet. Der Baum selbst enthält nur Schlüsselwerte

Hashtabellen

m = Tabellenplätze, **n**=Anzahl der Datensätze, $B = \frac{n}{m}$

Hashfunktionen

Multiplikation: $h(s) \lfloor m\{sc\} \rfloor$ mit $\{sc\} = sc - \lfloor sc \rfloor$; optimal mit $c = 0.5(1 + \sqrt{5})$

Implementieren mit Shift-Operationen für $m = 2^p$, $p \leq$ wortbreite

=> die p vordersten Bits des unteren Worts von $s \cdot c$ (low-Register)

Bsp: $w=8$, $p=6$, $c=0.618=0.10011110$, $s=4=100$

$h(s) : 10011110 * 100 = 1001111000 \Rightarrow h(4) = 30$

Universelle Familien:

Zufallsfunktion= Zufällig Wert->Hash Kombination, Kollisionswahrscheinlichkeit: $1/m$

Funktionsfamilie ist universelle Familie, wenn Kollisionswahrscheinlichkeit $= \frac{1}{m}$

Bsp: Primzahl p, Tabellengröße

$m \leq p$; Wähle $a, b \in \mathbb{Z}_p$

$h(x) = ((ax + b) \bmod p) \bmod m$

Bsp: Primzahl p, $a = (a_1, \dots, a_r) \in \mathbb{Z}_p^r$, x als p-adische Entwicklung, $x \leq p^r - 1$

$h_a(x_1, x_2, \dots, x_r) = \sum_{i=1}^r a_i x_i \bmod p$

Verkettung mit Überlaufbereich

Hashfunktionen liefern Adressen im Primärbereich, Tabelleneinträge speichern zusätzlich

Nachfolgeradresse im Überlaufbereich; Freie überlaufzellen zusätzliche verkettung

Zugriffe bei Erfolg: $< 1 + \frac{1}{2}B$, Zugriffe bei Misserfolg: $\leq 1 + B$

Wahrscheinlichkeit für i Kollisionen: $p_i = \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{n-i}$

=> Überlaufbereich $= n - m(1 - p_0) =$ Kollisionen (m =Primärbereich)

Freie Plätze im Mittel: $\lfloor (n - m - \text{Überlauf}) \rfloor$

Offene Adressierung

Bei Kollision wird ein anderer Platz in der Tabelle gesucht, anhand der Sondierfolge alle Plätze müssen in Folge $i(s)_j$ enthalten sein).

Einfügen: Suche erste freie/gelöschte Zelle in Sondierfolge, füge ein;

Suchen: Durchlaufe Sondierfolge, bis gefunden oder sicher nicht in Tabelle

Löschen: Suche und markiere als gelöscht

mittlere Länge Sondierfolge beim Suchen: $\frac{1}{B} \ln \left(\frac{1}{1-B} \right)$ beim Einfügen: $1/(1-B)$

Lineares Sondieren: betrachte immer den nächsten eintrag bis Erfolg; $(i(s)_j = h(s) + j \bmod m)$

Nachteil: Sondierfolgen verketteten sich (Cluster)

mittlere Länge Sondierfolge beim Suchen: $\frac{1}{2} \left(\frac{1}{1+B} \right)$ beim Einfügen: $\frac{1}{2} \left(1 + \left(\frac{1}{1-B} \right)^2 \right)$

Quadratisches Sondieren: $m = \text{Prim}; m \equiv 3 \bmod 4$

$i(s)_j = h(s) \pm j^2 \bmod m$ (also 0,+1,-1,+4,-4,...)

Doppelhashing: $i(s)_j = h(s) + jh^*(s) \bmod m$, $h \neq h^*$, $m = \text{prim}$

Graphen

k = Anzahl Knoten, **e** = Anzahl Kanten

a adjazent zu b: es existiert die kante $a \rightarrow b$, also $(a, b) \in E$

Umgebung: alle zu einem knoten adjazenten knoten

$e \leq \binom{k}{2}$ (ungerichtet) bzw. $\leq k(k-1)$ (gerichtet)

Teilgraph: Knoten und Kanten sind Teilmenge des Originalgraphen

aufspannender Teilgraph: alle Knoten und Teilmenge der Kanten des Originalgraphen

erzeugender Teilgraph: Teilmenge der Knoten und alle Kanten zwischen diesen

Pfad: Folge von Knoten v_0, v_1, \dots, v_n , mit Kanten von $v_i \rightarrow v_{i+1}$

Zyklus: geschlossener Pfad mit länge ≥ 3 (ungerichtet) bzw. ≥ 2 (gerichtet)

Zusammenhangskomponente: alle gegenseitig erreichbaren knoten bilden Komponente

Baum: zusammenhängend, azyklisch und $e = k-1$

Bipartiter Graph: zwei Mengen von Knoten V_1, V_2 , alle Kanten gehen von V_1 nach V_2

Adjazenzmatrix: $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \Rightarrow \begin{matrix} & 2 \\ 1 & \nearrow & \\ & \longleftarrow & 3 \end{matrix}$ Adjazenzliste: $\begin{matrix} 1: 2 \\ 2: \\ 3: 1 \end{matrix}$

Weitensuche

Besuche die Nachbarn des Startknotens, dann die Nachbarn des ersten Nachbarn usw.

\Leftrightarrow gehe den entstehenden Baum Ebenenweise durch.

V_T : besuchte Knoten, V_{ad} : zum besuch Vorgemerkte Knoten als Queue, V_R : Rest

Ablauf: $\text{int}[k] \text{ where; } // < 0: \text{ in der Queue, } 0: V_R, > 0: \text{ besucht}$

Visit(Node k): füge k in die Queue;

durchlaufe die Queue, für jeden Knoten füge alle Nachbarn aus V_R in die Queue

Laufzeit: bei Liste: $O(e+k)$, bei Matrix: $O(k^2)$

Erweiterung: Test auf Zyklen \Leftrightarrow Test ob Nachbar schon im Baum (und nicht parent)

Ermittlung des Abstands von der Wurzel des erzeugten Baums

Tiefensuche

Besuche den 1.Nachbarn des Startknotens, dann den 1.Nachbarn des 1.Nachbarn usw.

\Leftrightarrow durchlaufe einen Pfad nach dem Anderen

Visit: durchlaufe die Adjazenzliste, für jeden nicht besuchten nachbarn rufe Visit auf

Laufzeit: $O(e+k)$

Erweiterung: Test auf Azyklität: Kanten auf einen Vorgänger

bei Visit Start/Ende: A vorgänger von B wenn $[Start_B, End_B] \subset [Start_A, End_A]$

Topologisches Sortieren: Array der Länge k, fülle von hinten bei Visitende

Starke Zusammenhangskomponente::

1. Nummeriere in Terminierungsreihenfolge
2. Drehe alle Kanten um (Konstruiere den reversen Graph)
3. Tiefensuche von höchster Terminierungsnummer aus, alle Erreichbaren sind starke Zusammenhangskomp.
4. (wiederhole letzten Schritt bei bedarf)

Priority Queue

Speichert Elemente mit Prioritäten, entnahme des Elements mit kleinster Priorität;

Implementierung durch Heap und Positionsarray für die Elemente;

Wird durch UpHeap/DownHeap die Position verändert: anpassen des Positionsarrays

Einfügen: füge das Element am HeapEnde ein, Umgekehrtes DownHeap (UpHeap)

Löschen: Entnahme der Heapwurzel, ersetzen durch letztes Element; DownHeap

Union-Find

dynamische Partitionierung; parent array; Für Erweiterungen: array rank

Partition als Wurzelbaum => Elemente in einer Menge wenn gleiche Wurzel

Repräsentant ist Wurzel; Wurzeln haben parent[w] = 0

Find(e): return Wurzel der Partition von e : durchlaufe parent-Beziehung bis Wurzel

Pfadkomprimierung:: setze gefundene Wurzel als parent aller Knoten auf diesem Pfad

Union(x,y): return true wenn in selber Partition, sonst false + vereinige diese Partitionen

i = Find(i); j = Find(j) if(i!=j) { parent[i] = Find(j) }

Höhen-Balancierung: rank[i] speichert rang von i; hänge bei Union die Kleinere Wurzel

unter die größere, bei gleichheit steigt der rang der neuen Wurzel

worst-case Laufzeit für n-1 unions und m finds: $O((m+n)\alpha(n))$; $\alpha(n) \leq 4$

Dijkstra/Prim (minimaler aufspannender Baum)

Start:: ({Startknoten}, \emptyset) // Startknoten, keine Kanten

Schritt:: füge die kleinste vom konstruierten Baum ausgehende Kante in den baum ein;

Priorität: bei Prim: Kantengewicht, bei Dijkstra Pfad zur Wurzel

Implementierung durch Priority Queue bei Adjazenzliste

Laufzeit: $O(n^2)$ bei Matrix, $O((p+q)\log(p))$ bei liste

Kruskal (minimaler aufspannender Baum)

Start: T= (V, \emptyset) // alle Knoten, keine Kanten

Schritt: füge die kleinste Kante ein, die keinen Zyklus erzeugt

Implementierung: Knoten in Union-Find; Sortiere Kanten nach gewicht, durchlaufe die

Kanten und führe für jede Kante Union(v_{Start}, v_{End}) aus; wenn false füge die Kante ein;

Laufzeit: $O(p+q\log(q))$

Boruvka (minimaler aufspannender Baum)

Min-Max-Ordnung: kante ist Kleiner falls gewicht kleiner bzw. kleinerer Knoten kleiner

bzw. größerer Knoten kleiner

minimale indizente Kante: kleinste Kante an einem Knoten

Start:: Tree(V, \emptyset) Graph(V,E)

Schritt:: füge alle Minimal indizenten Kanten im Baum ein, Kontrahiere sie im Graph; wiederholen Laufzeit: $O((p+q)\log(p))$

Warshall (transitiver Abschluss)

Erzeugt Graph, bei dem jede Kante einem Pfad in der Eingabe darstellt

Start:: a_0 = Adjazenzmatrix;

für k=1 bis p:

Schritt:: $a_k[i, j] = a_{k-1}[i, j] \text{ or } (a_{k-1}[i, k] \text{ and } a_{k-1}[k, j])$ für implementierung: es wird nur speicher für eine Matrix benötigt, diese wird angepasst. 3 forschleifen (k,i,j) => $O(p^3)$

Floyd (minimaler aufspannender Baum)

Adjazenzmatrix gewichteter Graph, gewicht= ∞ wenn keine Kante, 0 in Hauptdiagonale;

Start:: a_0 = Adjazenzmatrix;

für k=1 bis p:

Schritt:: Schritt: $a_k[i, j] = \min(a_{k-1}[i, j], a_{k-1}[i, k] + a_{k-1}[k, j])$

funktioniert auch mit negativen gewichten, wenn keine negativen Zyklen, implementierung analog zu Warshall