O-Notation

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

Fester Wert => g =O(f(n)) UND f = O(g(n);

Differenzengleichungen

1. $x_1 = b,$
   $x_n = a_n x_{n-1} + b_n$
2. $\pi_n = \prod_{i=2}^{n} a_i, \pi_1 = 1$
3. $x_n = \pi_n \left(b + \sum_{i=2}^{n} \frac{b_i}{\pi_i}\right)$

Einfache Sonderfalle:

- $x_n = a_n x_{n-1} \leftrightarrow b\prod_{i=2}^{n} a_i$
- $x_n = x_{n-1} + b_n \leftrightarrow b + \sum_{i=2}^{n} b_i$

Nützliche Zahlen

- $\sum_{i=1}^{i} \frac{1}{n} = H_n \quad Abschatzung : ln(n+1) <= H_n <= ln(n)+1$
  $\sum_{i=1}^{n} H_i = (n+1)H_n - n$
- $\sum_{i=1}^{n} c = nc$
  $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(n+2)}{6}$
  $\sum_{i=1}^{n}(2i-1) = n^2 \qquad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{i=1}^{n} \lfloor log_2(i) \rfloor = (n+1)\lfloor log_2(n) \rfloor - 2\left(2^{\lfloor log_2(n) \rfloor} - 1\right)$
- $\sum_{i=1}^{n} x_i^i = \frac{x^{n+1}-1}{x-1}$
  $\sum_{i=1}^{n} i x_{i-1}^i = \frac{(n+1)x^n(x-1)-(x^{n+1}-1)}{(x-1)^2}$

rationale Summen

1. $\sum_{i=1}^{n} \frac{x_i}{polynom}$
2. $Partialbruchzerlegung : \frac{x_i}{polynom} = \frac{a}{1.NST} + \frac{b}{2.NST} \ldots$
   Bei Mehrfachen NST $(k.NST)^{Vielfachheit}$ statt k. NST
3. Koeffizientenvergleich;

Logarithmen

- $log(xy) = log(x) + log(y)$
- $log(x^c) = clog(x)$
- $e^{log(x)} = x$

Selection-Sort

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

Bubble-Sort

durchlaufe alle elemente und tausche element mit nachfolger wenn es kleiner ist als dieser; wiederhole bis das array sortiert ist. **O{n^2}**

Quicksort

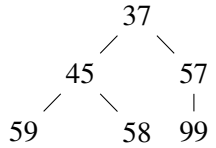
1. Wähle Pivot= letztes Element
2. lasse zeiger von beiden Enden des Restarrays nach innen laufen:
   wenn der rechte zeiger auf ein kleineres bzw. der linke auf ein größeres Element als das Pivot zeigt stoppe den zeiger; wenn beide stoppen tausche sie, wenn sich die Zeiger treffen tausche das Pivot nach innen
3. Wiederhole Quicksort im Rechten und linken Teilarray.

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

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

Heapsort

z.B. $[\underbrace{37}, \underbrace{45,57}, \underbrace{59,58,99}]=$



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 rechts nach links
Heapbedingung: Knoten ist $\leq$ seine Kinder

Einsickern: Knoten verletzt die Heapbedingung: Tausche mit Kleinerem nachfolger; ggf. wiederholen

Downheap Revisited: Folge dem Pfad der kleineren Nachfolger bis zum Blatt und speichere dessen index.

Vorgänger eines Blattes b_i ist $\lfloor b_i/2 \rfloor$ => index des i. Knotens sind die vordersten i Bits des Blattindex

Lineare Suche: Suche vom Pfadende aus die Stelle, an der die Wurzel eingefügt werden muss, schiebe alle Knoten um 1 Nach oben vom nachfolger der Wurzel angefangen.

binärsuche: analog zu lineare Suche nur mit binärer Suche

Aufbau des Heaps: Führe Einsichern für die Teilbäume durch (von unten rechts zur wurzel); **O(n)**

Heapsort: Erzeuge Heap; Tausche das letzte Element mit dem ersten, Wurzel einsickern in [1...n-1] , wiederhole bis array leer; Laufzeit worst-case: $O(n\log_2 n)$

Suche im Array

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

Binär: Vorraussetzung: Sortiertes Array; Wähle das mittlere Element; ist es kleiner als das zu suchende Wiederhole im linken Teilarray, analog bei größer
worst-Case: $O(\log_2 n)$

Quick-Select: ist das k. kleinste element gesucht. Analog zu probab. Quicksort, es muss allerdings immer nur das Teilarray sortiert werden, in den der k. Index fällt; durchschnitt **O(n)**

Bäume

Preorder: Knoten -linkesKind-rechtesKind

Inorder: linkesKind- Knoten -rechtesKind
Postorder: linkesKind-rechtesKind- Knoten

Binärer Suchbaum

jeder Knoten hat 2 Kinder, alle im linken Teilbaum sind Kleiner, alle im rechten größer

Suche: Starte in der Wurzel, wenn item größer gehe nach rechts, kleiner links, = gefunden; 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: |Balancefaktor|= 1

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

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

Einfügen/Löschen: wie bei binär + 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

Genau einmal Ausgleichen benötigt

Ausgleich nach Löschen:: (Umgekehrt für +2)

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

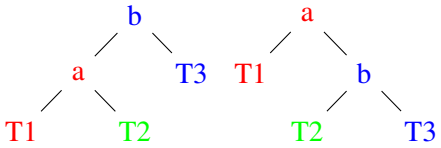
betrachte linken Nachfolger (a):

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.

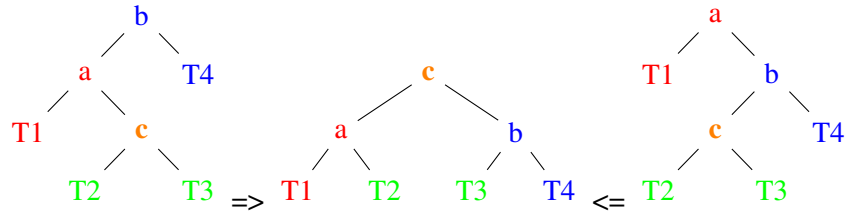
Rotationen



=> Rechtsrotation um a

<= Linksrotation um b

Doppelrotation



Treap

jedem element wird zusätzlich eine zufällige Priorität zugewiesen

Heapbedingung: das Elternelement hat kleinere Priorität als die Kinder

=> Treap ist eindeutig bestimmt

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

Erwartungswert Rotationen: <2

Suchen & Einfügen: Analog zu Binärer Baum; anschließend: Rotation des neuen Knotens nach oben, bis heapbedingung erfüllt ist; **Löschen**: Rotation des zu löschenden Knoten mit dem Nachfolger kleinerer Priorität, bis der Knoten ein Blatt ist, dann löschen.

B-Bäume

für festplatten, minimieren zugriffe in datenbanken

Ordung d: Knoten haben zwischen $\lceil \frac{d}{2} \rceil$ und d Nachfolger, leq $\lfloor \frac{d-1}{2} \rfloor - 1$ und d-1 Elemente,

Wurzel hat mind. 2 Nachfolger oder ist Blatt

alle Blätter sind immer auf einer Ebene => immer vollst. ausgeglichen

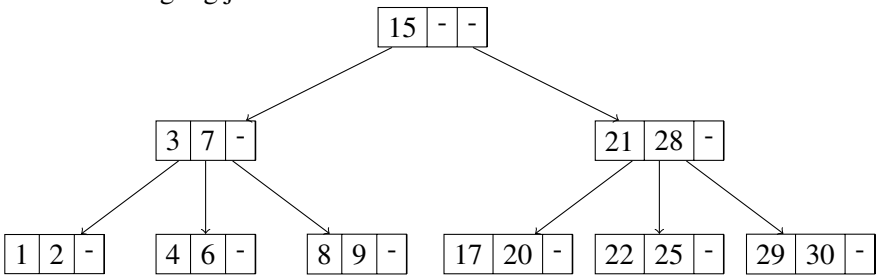
Baum mit höhe h hat mindestens $1 + 2\frac{\lceil \frac{d}{2} \rceil^h - 1}{\lceil \frac{d}{2} \rceil - 1}$ und maximal $\frac{d^{h+1}-1}{d-1}$ Knoten

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 jedes Element + rechte/linke Adresse



Bsp: d = 4

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

Wenn Blatt voll: (ggf. rekursiv)

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

Löschen:

1. Element nicht in einem Blatt => tausche es mit dem Nachfolger in der Sortierreihenfolge; Anschließend löschen;
2. ist die Seite danach zu Klein ($< \lfloor \frac{d-1}{2} \rfloor$) versuche Ausgleich mit direkten Nachbarblatt: das dazwischenliegende Element x wird aus Vaterknoten in den zu kleinen Knoten verschoben, der Nachfolger/Vorgänger aus dem anderen Knoten an seine Stelle eingefügt, sein Verweis kommt an die leere Adresse vor/nach x.
3. Ist das nicht möglich: Füge 2 benachbarte Knoten + x aus Vaterknoten zu einem Zusammen. Wiederhole ggf. rekursiv.

B*-Baum: Beim Einfügen in volle Seite wird wie beim löschen zuerst versucht mit direkten Nachbarn auszugleichen => bessere Speicherausnutzung, geringere Höhe

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

Hashtabellen

Speicherung in Tabelle, berechnung der Adresse durch Hashen eines Schlüssels

m = Tabellenplätze

Multiplikation: $h(s)\lfloor m\{sc\}\rfloor$ mit $\{x\} = x - \lfloor x \rfloor$; 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 = 10 | 01111000 => h(4)= 30

Universelle Familien: (theorie): Zufallsfunktion= Tabelle aller Zuordnungen Wert->Hash, zufällig gewählte Spalte als Hash-funktion; Kollisionswahrscheinlichkeit: 1/m

Funktionsfamilie ist universelle Familie, wenn Kollisionswahrscheinlichkeit = 1/m

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

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

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

$h_a(x_1, x_2, \ldots, x_r) = \Sigma_{i_1}^r a_i x_i\ mod\ p$

Kollisionsbehandlung

Verkettung mit Überlaufbereich

Hashfunktionen liefern Adressen im Primärbereich, Tabelleneinträge enthalten Nachfolgeradresse im Überlaufbereich; Freie überlaufzellen bilden zusätzliche verkettung

Zugriffe bei erfolgreicher Suche: $< 1 + 1/2B$, Zugriffe bei nicht erfolgreicher suche: $\leq 1 + B$

Kollisionswahrscheinlichkeit Verkettung: $p_i = \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{n-i}$

=> Überlaufbereich = $n - m(1 - p_0)$ = Kollisionen (m= größe Primärbereich, n=Anzahl Elemente)

Freie Plätze im Mittel: $-(n - m - berlauf)$

Offene Adressierung

Bei Kollision wird ein anderer Platz in der Tabelle gesucht, anhand einer 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

n von m Zellen belegt, B= freieZellen/alleZellen

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

Lineares Sondieren

Sondierfolge: direkt aufeinander folgende Tabelleneinträge $(i(s)_j = h(s) + j\ mod\ m$

Nachteil: Sondierfolgen verketten 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

Tabellenplätze m = Primzahl; idealerweise m = 3 mod 4

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

Doppelhashing

$i(s)_j = h(s) + jh^*(s)\ mod\ m$, idealerweise wahl von h und h* unabhängig, m prim

Graphen

im folgenden **k = Anzahl Knoten, e = Anzahl Kanten** a adjazent zu b: es existiert die kante a->b, also $(a,b) \in E$

Umgebung: alle zu einem knoten adjazenten knoten

q $\leq \binom{p}{2}$ (ungerichtet) bzw. $\leq p(p-1)$(gerichtet)

Pfad: Folge von Knoten $v_0, v_1, \ldots, v_n$, mit Kanten von $x_i -> x_{i+1}$

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

perfekte Zuordnung: Bipartiter Graph, bijektive Abbildung von V_1 nach $V_2 \Leftrightarrow$ det(Adjazenzmatrix) = 0

Adjazenzmatrix: $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ => Graph mit 3 Knoten, kante von 0->1

Adjazenzliste: Für jeden Knoten eine Liste der adjazenten Knoten.

Weitensuche

Besuche die Nachbarn des Startknotens, dann die Nachbarn des ersten Nachbarn usw.
⇔ gehe den entstehenden Baum Ebenenweise durch.
V_T : *besuchteKnoten*, V_{ad}: zum besuch Vorgemerkte Knoten als Queue, V_R: Rest

Ablauf: int[k] where; <0: in der Queue, 0: V_R, >0: besucht (nummer gibt die Zusammenhangskomponente an);
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)
Erweiterungen: Test auf Zyklen ⇔ 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.
⇔ durchlaufe die Pfade bis zum ende.
Visit: durchlaufe die Adjazenzliste, für jeden nicht besuchten nachbarn rufe Visit auf
Laufzeit: O(e+k)
Trägt man die Start/Ende-Zeitpunkte von Visit ein, dann ist a vorfahr von b wenn $[Start_b, End_b]$ in $[Start_a, End_a]$ liegt.
Erweiterung: Test auf Azyklität ⇔ Kanten auf einen Vorgänger
Topologisches Sortieren: Array mit Länge=|Knoten|, fülle das Array von hinten in Terminierungsreihenfolge
Starke Zusammenhangskomponente: 1. Nummeriere in Terminierunsreihenfolge, 2. Konstruiere den reversen Graph (Drehe alle Kanten um) 3. starte Tiefensuche bei Knoten mit höchster Terminierungsnummer, alle Erreichbaren sind starke Zusammenhangskomponente (wiederhole letzten Schritt bei bedarf)

Priority Queue

Speichert Elemente mit Prioritäten, abgreifen der Elemente in aufsteigender Reihenfolge der Elemente; methoden PQUpdate, PQRemove; Implementierung durch Heap und Positionsarray für die Elemente;
Wird durch UpHeap/DownHeap die Position verändert muss das Positionsarray angepasst werden;
Einfügen: füge das Element am HeapEnde ein, UpHeap
Löschen: Nimm das erste element aus dem Heap, tausche das letzte an die 1. Stelle und DownHeap Laufzeit: O(log(n))

Union-Find

dynamische Partitionierung einer Menge: speicherung in parent array; zusätzliches Array rank für Erweiterungen;
jede Partition wird ein Wurzelbaum => Elemente in einer Menge wenn gleiche Wurzel Repräsentant ist Wurzel; Wurzeln haben parent[w] =0
Find(e): liefert den Repräsentanten (Wurzel) der Partition in der e liegt zurück
durchlaufe parent-Beziehung bis zur Wurzel
Pfadkomprimierung: anschließend: setze parent aller Knoten auf dem Pfad von e -> wurzel auf die gefundene Wurzel
Union(x,y): true wenn in selber Partition, sonst false + vereinigen diese Partitionen
Zusammenfassen von i = Find(i) und j = Find(j) durch; parent[i] = Find(j)
Höhen-Balancierung: bei Union wird die kleinere Wurzel Kind der größeren; 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}, ∅)
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,∅) // 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+qlog(q))

Boruvka (minimaler aufspannender Baum)

Min-Max-Ordnung: kante ist Kleiner falls gewicht kleiner bzw. kleinster Knoten kleiner bzw. größter Knoten kleiner
minimale indizente Kante: kleinste Kante an einem Knoten
Start: Tree(V,∅) 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)

Transitiver Abschluss von G: Graph, bei dem Kante (a,b), wenn Pfad von a nach b in G (bei ungerichteten: Zusammenhangskomponente)
Start: a_0 = Adjazenzmatrix; für k=1 bis p: Schritt: $a_k[i,j] = a_{k-1} or (a_{k-1}[i,k] and a_{k-1}[k,j])$

für implementierung: es wird nur speicher für eine Matrix benötigt, diese wird angepasst.
3 forschleifen (k,i,j) $\mathbf{O}(p^3)$

Floyd

Adjazenzmatrix gewichteter Graph, gewicht= ∞ wenn keine Kante, 0 in Hauptdiagonale; Start: a_0 = Adjazenzmatrix; für k=1 bis p: Schritt: Schritt: $a_k[i,j] = mina_{k-1}, a_{k-1}[i,k] + a_{k-1}[k,j]$ funktioniert auch mit negativen gewichten, wenn keine negativen Zyklen implementierung analog zu Warshall