repetition

Dienstag, 9. August 2016
13:46

Bubblesort
$n^2\ Vergleiche, n^2\ Verschiebungen$
in situ, stable
Selectionsort
$n^2\ Vergleiche, n\ Verschiebungen$
in situ, nicht stable
Insertionsort
$bin\ddot{a}re\ Suche\ * n = nlog(n) Vergleiche, n^2\ Verschiebungen$
in situ, stable
Mergesort
$nlog(n)\ Vergleiche, nlog(n) Verschiebungen$
nicht in situ, stable
Heapsort
$Versickern: \log(n) * 2\ Vergleiche$
$Es\ wird\ n\ Mal\ versickert: nlog(n) * 2$
$Heap\ bauen\ kostet \frac{n}{2} * 2\log(n) \Rightarrow O\big(2nlog(n)\big) Vergleiche\ insgesamt$
Im schnitt sind es $O(nlog(n) + n) Vergleiche$
$O\big(nlog(n)\big) Verschiebungen$
in situ, nicht stable
Bottom-Up Heapsort
Finde Pfad fürs Versickern mit log(n)
Finde Ort auf dem Versickerpfad mit binärer Suche mit log(log(n))
also gilt O(log(n)+log(log(n)) Vergleiche und O(nlog(n)) Verschiebungen
in situ, nicht stable
Natürliches Mergesort
n Vergleiche in $\log(k)$ steps => $O(nlog(k)$ Vergleiche und Verschiebungen wenn am Anfang k sortierte Stücke vorlagen.
nicht in situ, stable
Quicksort
Teile mit Median von Blum in zwei hälften. (pivot wird gerne auch als das ganz rechts gewählt. Am schluss wird das pivot hineingetauscht nachdem sich die pointer geschnitten haben)
Gehe von links und rechts mit pointern durch und vertausche items falls beide auf falscher seite.
repeat log(n) times
Anzahl Vergleiche: $O(n * \log(n))\ im\ mittel\ und\ Best\ Case, aber\ O(n^2) im\ worst\ case$
Anzahl Verschiebungen: $O(nlogn), im\ BestCase\ keine$
nicht stable und in situ
Bucketsort
stable und in situ, O(n)
Radixsort MSD
Bucketsort repetiert bis fertig. O((Zahlen+Ziffernmöglichkeiten)*länge)
Radixsort LSD
Zuerst nach der kleinsten, dann jeweils in gruppen der grösseren ziffer einsortieren.
auch in O(Zahlen+Ziffernmöglichkeiten)*länge)

AVL baum
Rotation: Von unten nach oben durchgehen. Wenn das neue Element bzw der unausgeglichene Baumteil auf der innenseite ist, sodass normales rotieren einfach nur das problem nach rechts verschieben würde, wird zuerst der Teilbaum rotiert. Bsp für rechtsrotation: das Element r, das die Wurzel des rechten teilbaums von a ist, wird zum parent von a und das ursprüngliche parent von a rutscht rechts runter.

Splay Tree
Element wenn abgefragt durch zig, zag zigzig und zigzag zur wurzel bringen.
Falls es um zig-zig oder zag-zag geht wird an dem Wurzelnahen Element begonnen, dann bei fernenen. Bei zig-zag oder zag-zig wird zuerst das kind bewegt. Wird gelöscht, so wird der symmetrische vorgänger zur wurzel bewegt, wonach das Element bequem gelöscht werden kann.

Huffman coding
Pair lowest frequency up, set their root to the sum.
Continue until all single keys given away. Repeat until only one root left.
Now encode: left is 0

Optimaler Suchbaum
1. Keys und Intervalle notieren
2. W auf diagonale = Weight(key) bzw Weight(interval)
3. W über diagonale = W(i,j-1)+W(j+1,k)+W(j,j)
4. P auf diagonale = 0
5. P über diagonale = W(über diagonale)
6. P darüber = W(j,j)+min(P(i,j-1)+P(j,k))
7. R durch rückverfolgung oder nebenbei notieren. Falls nebenbei notieren gilt r = l von vorher
8. Auslesen: von oben rechts her. Das ganz oben rechts ist die erste root.

3,4,5B-Tree
Jede Node hat 5/2 aufgerundet = 3 Kinder minimal. 5 Maximal
Jede Node hat children - 1 keys (?)
Ausbalancieren wenn limit überschritten: Mittleren key behalten als root, davon kinder mit den restlichen keys.
Die Wurzel hat beliebig wenig kinder

Branch and bound
Wie backtracking, aber eine globale untere und eine lokale obere Schranke festlegen. Wird die globale untere Schranke unterschritten, kann dieser Teil der lösungen ignoriert werden. Im allgemeinen wird zuerst der Teil mit der höchsten oberen Schranke berechnet und das setzt die globale untere schranke neu, wenn man nach der höchsten Lösung sucht.

Kruskal vs Prim
Kruskal wählt die kürzeste Kante, die noch nicht gewählt wurde. Prim wählt die kürzeste **adjazente** Kante die noch nicht gewählt wurde. Kruskal schliesst daraufhin alle kanten aus, die einen Kreis bilden würden. Kruskal ist am besten mit Union-Find-Struktur (mit anfänglich jedem Punkt als eigenem Baum und beim Kante wählen die bäume vereinigen. Falls sie schon vereinigt sind, discard die kante.) Prim läuft am besten mit fibonacciheap mit $O(|E| + Vlog(V))$
Kruskal hat Laufzeit $O(E * \log(E)) bzw\ mit\ UnionFind\ O(E * [\log^* E \approx 1]) \approx O(E)$

Union
Der kleinere baum wird an den grösseren angehängt. bei Union(r,s) wird r, falls es noch nicht die Wurzel ist, mit s vertauscht.

Kürzester Weg nach Dijkstra
Beschrifte alle angrenzenden Punkte mit ihrem min Wert (Rekursiv?). Dann wähle den Punkt mit dem kleinsten Weg und streiche ihn. Wiederhole mit den nun adjazenten Punkten. Keine negative kanten.
Laufzeit $\Theta\big((E + V) * \log(V)\big)$
Mit Fibonacci heap: $O((E + Vlog(V))$

Kürzester Weg von s nach alle Punkte nach Bellman-Ford
für alle (u,v) Paare, geht auch mit negativen kanten
Falls Distanz(s,u)+Distanz(u,v)<Distanz(v), setze Distanz(v) auf Distanz(s,u)+Distanz(u,v)
Der Pfad nach v geht also via u.

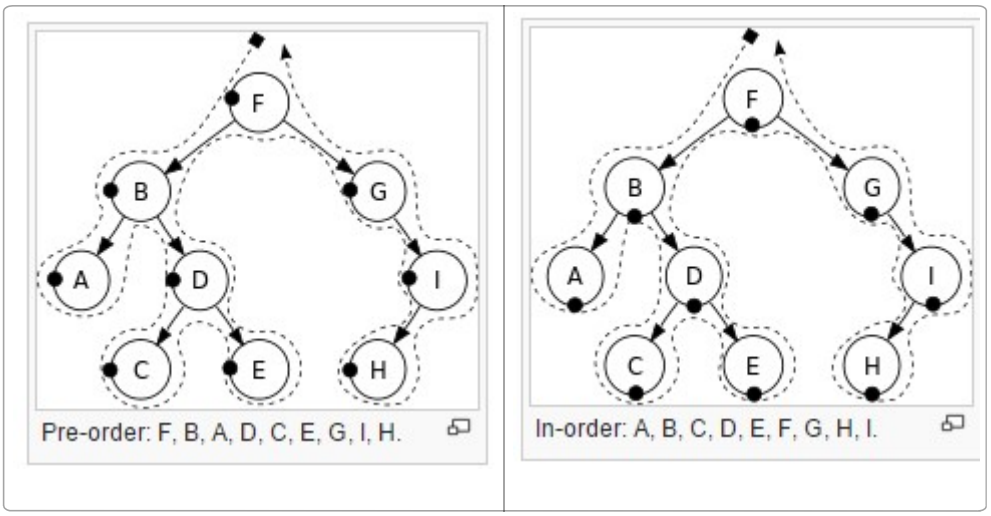
Anfänglich alles auf ∞ initialisieren.
Wiederhole V-1 Mal. Wenn es immer noch ein (u,v)Paar gibt, sodass die Bedingung erfüllt ist, muss es einen negativen Zyklus geben, abbrechen.
Laufzeit $O(V * V)$
Mit Fibonacciheap $O(\log(V) * V + E)$

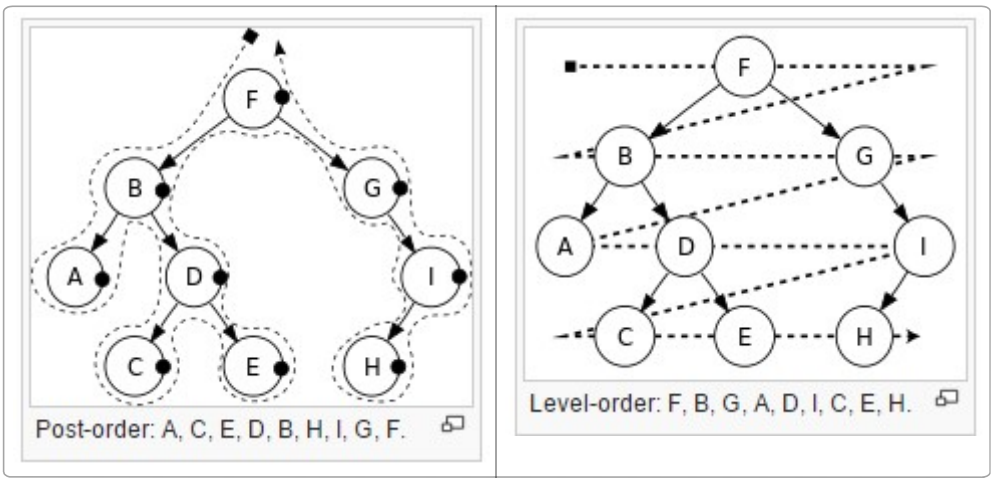
In jedem Schritt werden punkte aktualisiert, die in maximal i kanten erreicht werden können. bei i = V-1 ist eigentlich fertig. Repeat zum auf zyklen mit negativ testen.

MaxFlow/MinCut nach Ford & Fulkerson von s nach t mit gewichteten Kanten
1. wähle beliebigen Pfad
2. ziehe ihn ab und verlege ihn rückwärts
3. wiederhole bis kein Pfad mehr übrig

Laufzeiten
$O(n\log(n)) \in O(\sqrt{n})$
$O(\log(n)^k) \approx O(\log(n))$
$O\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \in O(n!) \in O(n^n)$
$O\binom{n}{k} \in O(n^k)$
$O(n^c) \in O\left(\frac{n^k}{\log(n)}\right), k > c$

Order





Fibonacci-heap

Hat immer einen Pointer zum minimalen Element. Der Heap ist eine Liste von Bäumen. Jede Wurzel hat mindestens die Priorität ihrer Kinder als Schlüssel.
Höchste Priorität = kleinster Key (minheaps)
einfügen: das Element als neuen Baum in die Liste legen. Pointer für MinElem ggf aktualisieren.
*merge:*Listen der zu mergenden Bäume werden verkettet. Pointer ggf aktualisieren.
*decreaseKey:*ändere Schlüssel. Falls die node ≠ Wurzel, markiere sie. Falls markiert, entferne ihren Vater aus dem baum und beginne damit einen neuen. Dadurch ist die Anazhl Knoten in einem Baum wo die Wurzel k Kinder hat auf die k+2te fibonaccizahl nach unten beschränkt.
extractMin:

1. Gib minimum aus
2. füge alle kinder vom minimum als Bäume zum Heap hinzu
3. lösche minimum
4. cleanup

cleanup:

1. Array der länge 0 bis 2log(n)
2. bei i ist ein Pointer zu einem Baum mit grad i
   gibt es mehrere Bäume mit Grad i, so wird das Element mit der höheren Priorität zum Vater des anderen. Dadurch erhöht sich der Grad des Baumes um 1 und er wird stattdessen bei i+1 eingefügt.
3. Es hat maximal 2log(n) elemente in diesem Array => $O(\log(n))$

remove:

1. decreaseKey bis es das neue minimum ist
2. extractMin

=> $O(\log(n))$

Man muss den von insert returnten Pointer erinnern. Suchen ist nicht effizient möglich.

Binomialheaps

Effizientes mergen: Füge heap mit grösserer wurzel dem anderen als kind der wurzel hinzu (minheap) falls anzahl bäume mit ordnung k = 2. Falls 1 einfach in heap übernehmen, falls 0 nichts machen, falls 3 wird der Baum aus dem übertrag in den heap aufgenommen und die anderen beiden gemerget zu einem der ordnung eines höher, der dann übertragen wird - d.h. der grössere-wurzel-baum ist das linkste kind des kleineren-wurzel-baums.
Ordnung 3 heisst, die wurzel hat 3 kinder.