Algorithmen & Datenstrukturen Zusammenfassung

Dynamische Konnektivität

Merge N Objekte

Union (a,b) → verbindet zwei Objekte

(a,b) connected → gibt es einen Weg über den a,b verbunden sind?

reflexiv → p ist mit p verbunden

Symetrisch → p ist mit q und q mit p verbunden

transitiv → p ist mit q und q mit r verbunden
daraus folgt p ist mit r verbunden

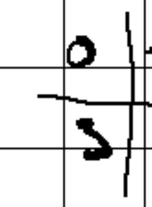
geschrieben als Zusammenhangskomponente

{1}, {2,3,4}, {5,6,7,8}

Quick Find

→ array indem die Verbindung stehen

id	0	1	2	3
i[id]	3	3	2	3

→ 

⇒ id → objekte
i[id] → verbunden mit

überprüfung ob zwei verbunden sind (i[p] == i[q])

↳ bei Union müssten jedoch alle objekte mit p als i[id] geändert werden.

⇒ Überprüfung O(1)

Union $O(N^2)$

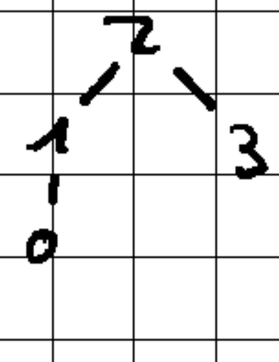
Quick Union

Array ist Baumförmig aufgebaut

↳ Elemente hängen nicht alle an der gleichen Wurzel aber die Wurzeln sind verbunden

i	0	1	2	3
id[i]	1	2	2	2

⇒



⇒ Union wird schneller, da man nur Wurzeln verbinden muss

⇒ $O(N)$

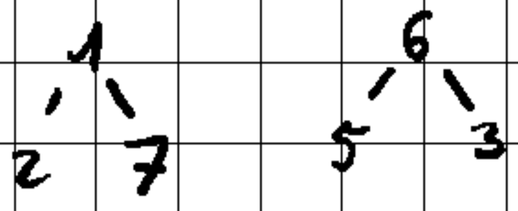
⇒ Find geht maximal $O(N)$

↳ ist abhängig von der Tiefe, weil immer die Wurzel auf neue Wurzel geprüft werden muss.

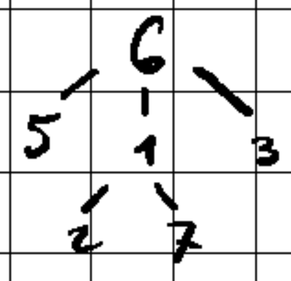
Weighted Quick Union

→ Quick Union, jedoch wird die tiefe von jeden Branch berücksichtigt und immer der kleinere an den grösseren gehängt wird.

⇒



Union(2,6) ⇒



Maximale Tiefe: $\lfloor \log_2 N \rfloor$ Anzahl Elemente

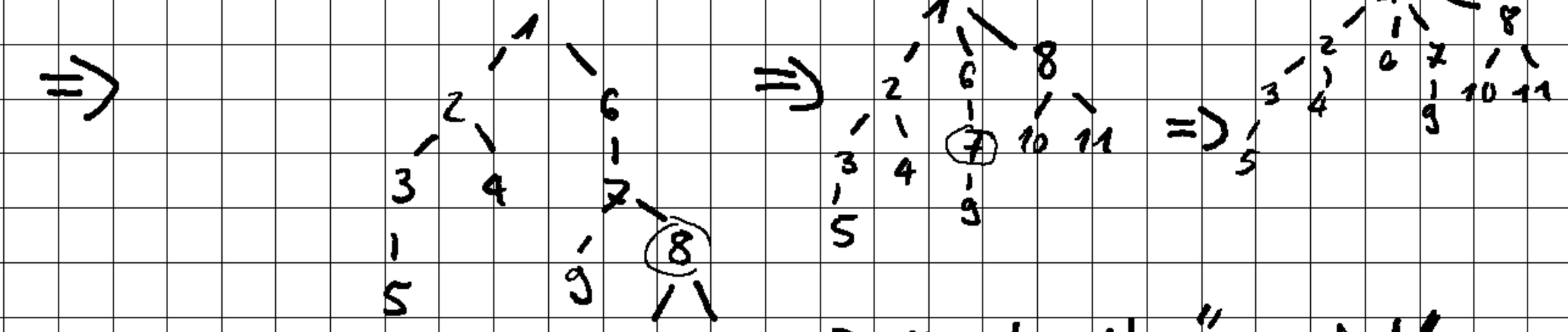
⇒ Union $O(\log_2 N)$

Find $O(\log_2 N)$

Pathcompression

→ die Wurzeln zeigen auf die Ur-Wurzel

Achtung: rekursiv nach dem die "unterste" Wurzel gefunden ist



⇒ Wurzel geht "eins hoch"

(start bei wurzel von 11)

Überblicke

weighted Quick Union mit Pathcompression (WQUPC)

M operations ⇒ $O(N + M \lg^* N)$

(Union-Find)

⇒ Quick-Find : $M \cdot N$

Quick-Union : $M \cdot N$

Weighted Quick Union : $N + M \cdot \log N$

Quick-Union & Pathcompression : $N + M \cdot \log N$

WQUPC : $N + M \cdot \lg^* N$

↳ $\lg^*$ = Anzahl Schritte bis < 1

⇒ $\approx O(1)$

Analyse von Algorithmen

→ Zuerst implementieren, dann analysieren

↳ Beobachten, Hypothese aufstellen

↳ solange wiederholen, bis Hypothese = Beobachtung

Empirische Analyse

→ Algorithmus mit verschieden grossen Datensätzen laufen lassen und die Zeit messen.

↳ $\log_2$ $\log_2$ Graph zeichnen $(\log_2 T(N), \log_2 N)$ (Laufzeit)

↳ alle Punkte sind nun gegeben als $a \cdot N^b$ $(a = 2^c)$

$\Rightarrow 2^c N^b \Rightarrow \log_2(T(N)) = b \cdot \log_2(N) + c$

$\Rightarrow$ Anhand dessen kann die Steigung berechnet werden

Mathematische Analyse

→ Gesamtlaufzeit = Kosten · Häufigkeit der Operationen

$\Rightarrow$ jede Operation braucht $\approx$ nano sekunden für die Ausführung

dabei ist zu beachten, dass die Geschwindigkeit jeder Operation noch von der Programmiersprache und vom PC abhängig ist, also ist

$T(N) = C_1 \cdot \text{Operation}_1 + C_2 \cdot \text{Operation}_2 \ldots$

Operation	Häufigkeit
Variabeldeklaration	$N+2$
Zuweisung	$N+2$
kleiner als Vergleich	$\frac{1}{2}(N+1)(N+2)$
ist gleich Vergleich	$\frac{1}{2}N(N-1)$
Array Zugriff	$N(N-1)$
Inkrement	$\frac{1}{2}N(N-1)$ bis $N(N-1)$

⇒ Zur Vereinfachung kann man auch nur die Array Zugriffe berücksichtigen.

Laufzeit Notationen

Big-O: $O(N)$

gibt die Obergrenze einer Funktion

⇒ $\frac{1}{6}N^3 + 2N^2 \Rightarrow O(N^3)$

Tilde: $\sim N$

gibt eine Annäherung

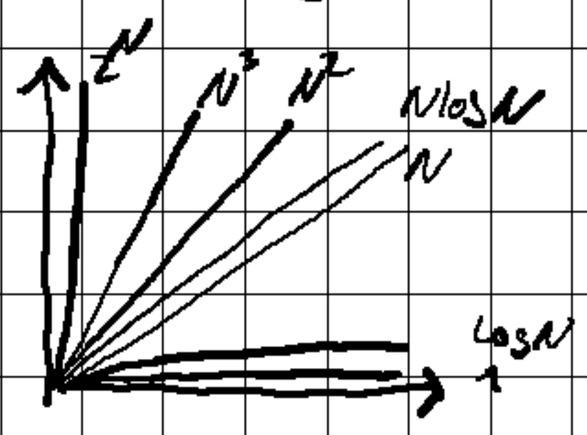
→ vernachlässigt Terme niedriger Ordnung

⇒ $\frac{1}{6}N^3 + 2N^2 \Rightarrow \sim \frac{1}{6}N^3$

Order-of-Growth

typische Laufzeiten: const logarithmisch linear linearithmetisch quad. kubisch. expo.
$1, \log N, N, N \log N, N^2, N^3, 2^N$

im $\log_2 \log_2$ Plot:



⇒ Ziel ist es einen logarithmischen Algorithmus für ein Problem zu finden

Input Abhängig

→ Laufzeiten ändern sich bei voschiedenem Input wir untersueten

Best Case: einfachster Input (Untergrenze)

Worst Case: schwirigster Input (Obergrenze)

Average Case: Zufällige Input (durchschnitt)

Speicherverbrauch

→ alle Daten im Code brauchen gewissen Speicherplatz

Byte = 8 Bits

Java Kostenmodell:

Typ	Byte
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

Einfache Typ

Typ	Byte
char[]	2N+24
int[]	4N+24
double[]	8N+24

1D Arrays

Typ	Byte
char[][]	~ 2MN
int[][]	~ 4MN
double[][]	~ 8MN

2D Arrays

Objekt in Java:

Object Overhead 16 Byte

Referenz (z.B Array) 8 Byte ⇒ (z.B int[]x; //new Referenz
deklaration → x= new int[N])

Padding

Total vielfaches von 8 (gefüllt mit Padding)

⇒ Zwei Methoden zur Analyse:

Deep Memory usage — zählt Referenz

Shallow Memory usage — ignoriert Referenz

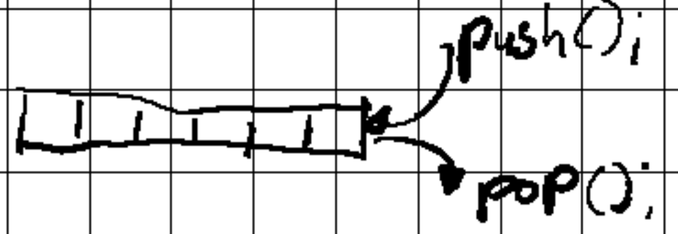
Datenstrukturen

→ logische Anordnung von Daten

- implementierung: Code der die Operationen implementiert
- Interface: Beschreibung von Datentypen & Basisoperationen
- Client: Programm das die Operationen verwendet die im Interface definiert sind

Stacks

→ Last-in - First-out

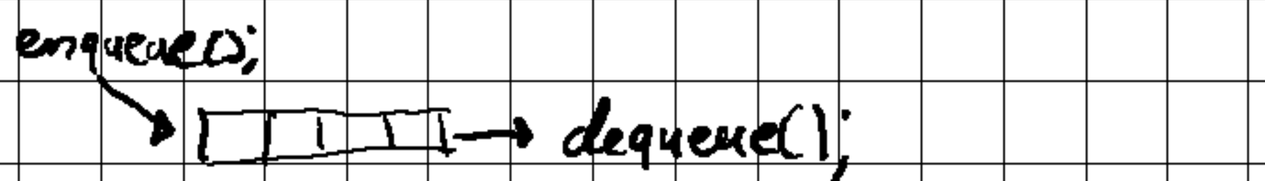


↳ Implementation

- LinkedList: worst Case → konstante Zeit
  Grösse → ~ 40 N Bytes
- Arrays: fixe Anzahl an Elementen
  Es können auch null Elemente eingefügt werden
- Dynamic Arrays: halbiert den Array wenn er nur $\frac{1}{4}$ gefüllt
  verdoppelt den Array wenn voll
  Laufzeit worst = N
  Best = 1
  Grösse = 8N bis 32N

Queue

→ First - in - last - out



implementierung

- LinkedList : Hinteres Element geht → Pointer läuft rückwärts
- Array :
- DynamicArray :

Priority Queue

→ Queue aber mit priorities

implementierung

→ sorted or unsorted

1. prio / 1 out

Search for 1. prio before out

Bäume

→ Datenstruktur zw darstellung hierarchischer Beziehungen

↳ Wurzel-Knoten die über Kanten mit Unterknoten verbunden sind
parent children

Arten:

- binäre Bäume: eine Wurzel hat maximal zwei Unterknoten
- balancierte Bäume: Baum der die Struktur ändert um eine möglichst geringe Tiefe zu haben
  ↳ garantiert O(log N) Laufzeit

Heaps

→ eine Art von binärem Baum

↳ Max Heap: Wurzel ist immer grösser wie kinder

↳ Min Heap: Wurzel ist immer kleiner wie kinder

Einfügen:

- Element am Ende einfügen und solange mit der Wurzel tauschen bis Heap-Eigenschaft erfüllt ist. (swim)

Löschen:

- niedrigstes Element wird mit der Wurzel getauscht und dann wird es so lange nach unten verschoben, bis die Heap-Eigenschaft erfüllt ist (sink)

Generics

→ Type Placeholder <K> (make a variable Type)

↳ Funktioniert für Klassen, Methoden, Interfaces ....

Vorteile

- Type Security → absolut typenprüfung → fehler bei kompilierung statt während Laufzeit
- Modularität → Code reuse
- more readable → es ist klar mit welchen Type gearbeitet wird

Iterables

→ variable die durch Liste laufen kann und Element zurückgibt

⇒ Array kann in Liste konvertiert werden:

```
List<K> Listname = Arrays.asList(arrayName);

Iterator<K> xy = Listname.iterator();
```

Methods:

next(); → returns current value and moves +1

hasNext(); → true, wenn +1 auch ein Element ist, sonst false

Sortieralgorithmen

Insetion Sort

Funktionsweise

- Array besteht aus zwei Teilen: sortiert: unsortiert
- nehme das 1. Element von der unsortierten Seite und lege es an die richtige Stelle des sortierten Tels.
- wiederhole bis unsortiert. length = 0

Laufzeit

Best Case: $O(N)$ → array ist bereits sortiert → ein check pro "Feld"

Worst Case: $O(N^2)$ → array ist umgekehrte Reihenfolge

Average Case: $O(N^2)$ → alle Elemente müssen bewegt werden

Eigenschaften

- stabil → Elemente behalten ihre relative Reihenfolge
  (Z.B Personen sind A-Z sortiert und wir sortieren nach Alter → prio 1 Alter, prio 2 Alphabet)
- in Place → Der Speicherplatz, benötigt für die Sortierung ist konstant
- online → Es können Elemente während dem Sortieren eingefügt werden.

Selection Sort

Funktionsweise

- Beginne mit dem ersten Element der Liste
- suche das kleinste Element im unsortierten Teil
- Tausche diese zwei Elemente
- wiederhole bis sortiert

Laufzeit

Best Case: $O(N^2)$ → Liste ist bereits sortiert. Trotzdem muss die Liste für jedes Element geprüft werden

worst Case: $O(N^2)$ → Liste ist verkehrt sortiert. Liste muss für jedes Element geprüft werden

Average Case: $O(N^2)$ → Für jedes Element muss jedes geprüft werden.

Eigenschaften

nicht stabil → relative Sortierung kann sich verändern

in place → braucht keine andere Speicherstrukturen

nicht online → es können keine neuen Elemente während der Laufzeit hinzugefügt werden.

Shell Sort

Funktionsweise

- definiere eine Zahl x
- Vergleiche alle Elemente mit Abstand x zueinander und vertausche diese wenn nötig.
- reduziere die Zahl x
- wiederhole bis x=1

Laufzeit

Best Case: $O(N \log N)$ → bereits sortierter Array

Worst Case: $O(N^2)$ → rückwärts sortiert

Average Case: $O(N \log N)$ → Abhängig von der Spaltungssequenz

Eigenschaften

nicht Stabil → Elemente können ihre relative Sortierung verändern

in Place → benötigt keine zusätzlichen Datenstrukturen

nicht online → es können keine neuen Elemente während der Ausführung eingefügt werden.

Adaptiv → die Laufzeit ist schneller, je mehr der Array bereits vorsortiert ist

Spaltungssequenz → die Laufzeit hängt von der Sequenz ab, in der der Array gespalten wird
↳ beste Sequenz: $3x+1$

Merge Sort

Funktionsweise

Divide an conquer

- Teile den Array rekursiv in zwei gleich grosse Hälften, bis die Teile nicht mehr teilbar sind
- Sortiere alle Teile indem sie in der richtigen Reihenfolge wieder zusammengefügt werden (merge)

Laufzeit

Best Case: $O(N \log N)$ → Array ist bereits sortiert
↳ Array wird gesplittet und wieder zsm. gefügt

Worst Case: $O(N \log N)$ → Array ist umgekehrt sortiert

Average Case: $O(N \log N)$

Eigenschaften

stabil → verändert die relative Sortierung nicht
↳ bei zwei gleichen Elementen wird das linke gewählt

nicht in-place → benötigt zusätzlichen Speicherplatz für die Teilsequenzen

nicht online → es können keine Elemente während dem Sortieren hinzugefügt werden.

Parallelisierung → Die Teilsequenzen können unabhängig sortiert werden und somit kann man die Prozesse parallel ausführen

Quick Sort

Funktionsweise

- wähle ein Pivot element
- ordne den Array an, sodass alle Elemente die kleiner sind als das Pivot Element auf der linken und alle die grösser sind auf der rechten Seite sind.
- Führe die oberen Schritte rekursiv aus, bis der Array sortiert ist.

Laufzeit

Best Case: $O(N \log N)$ → wenn der Pivot immer das mittlere Element ist

Worst Case: $O(N^2)$ → Array ist umgekehrt sortiert und der Pivot wird immer das grösste/kleinste Element ist.

Average Case: $O(N \log N)$

Eigenschaften

nicht stabil → relative Sortierung kann je nach Pivot ändern

in Place → es werden keine zusätzlichen Speicher benötigt

nicht online → es können keine Elemente während dem sortieren eingefügt werden

parallelisierbar → rekursive Aufrufe können parallel ausgeführt werden.

Heap Sort

Funktionsweise

- Array in einen Heapify Max-Heap konvertieren
- Tausche das Oberste und Unterste Element im Heap
- Entferne das Unterste Element
- Heapify den verbleibenden Heap, bis er wieder ein Max-Heap ist
- wiederhole bis Heap leer ist.

Laufzeit

Best Case: $O(N \log N)$
Worst Case: $O(N \log N)$
Average Case: $O(N \log N)$

} muss jedes mal wieder Heap erstellen: $O(N)$ und dann elemente entfernen: $O(\log N)$

Eigenschaften

nicht stabil → relative Sortierung kann sich bei Heapify ändern

in place → benötigt keine zusätzlichen Datenstrukturen

nicht online → es können keine Elemente während dem Sortieren eingefügt werden.

Verbesserung von Sortier Algorithmen

Shuffle

→ mische den Array vor dem Sortieren, um immer den Avrage Case zu erreichen (meistens)

- z.B weise jedem Element eine zufällige Zahl zu und sortiere sie nach dieser Zahl

Cut-off

→ geignet für Sortier Algorithmen die Rekursiv sind

- definiere eine minimale Arraygrösse und verwende einen anderen sortier Algorithmus, wenn die Arraygrösse darunter fällt.
- verwende einen Algorithmus, der für kleine Inputs gut ist. z.B Insertion Sort.

Median-of-Three

→ geignet für Quick Sort

- wähle drei Elemente aus dem Array (erstes, mittleres, letztes) und definiere dass Median Element (mittleres Element) als Pivot

Positions bedingt grössen bedingt

Such Algorithmen

Sequenzielle Suche (Lineare Suche)

Funktionsweise

- gehe jedes Element im Array durch um das gesuchte zu finden

Laufzeit

Suchen:
- Average Case: $O(\frac{N}{2})$ → Array muss nur bis zur Hälfte überprüft werden
- Worst Case: $O(N)$ → Element ist zu hinterst

Einfügen:
- Immer: $O(N)$ → Es muss nur überprüft werden ob das Element bereits vorhanden ist

Löschen:
- Average Case: $O(\frac{N}{2})$ → Suchen, dann löschen
- Worst Case: $O(N)$ → suchen, dann löschen

Eigenschaften

- Funktioniert bei unsortierten Arrays

Binäre Suche

Funktionsweise

→ Array muss vor Sortiert sein

- nimm das mittlere Element im Array
- Wenn das gesuchte Element kleiner ist, wiederhole mit dem linken Teil des Arrays
- Wenn das gesuchte Element grösser ist, wiederhole mit dem rechten Teil des Arrays
- Wenn das gesuchte Element = mittleres Element → fertig

Laufzeit

Average Case: $O(\log N)$ → Array wird durchlaufen bis gefunden

Worst Case: $O(\log N)$ → Array wird komplett durchlaufen

Eigenschaften

Effizient → bei grossen Datensätzen, da Suchbereich immer halbiert wird

⇒ Array muss vorsortiert sein

Binäer Such Baum (BST)

→ Binäe Suche in binärem Baum

Funktionsweise

Suche

- Vergleiche den gesuchten Wert mit dem obersten knoten
- Ist er grösser, dann gehe zum rechten Unterknoten, ist er kleiner, gehe zum linken Unterknoten
- Wiederhole, bis Knoten = gesuchter Wert oder bis es kein Unterknoten mehr gibt

Einfügen → Suche nach Element → fehlender kroten einfügen

Löschen

keine Unterknoten → einfach löschen

ein Unterknoten → knoten löschen und durch Unterknoten ersetzen

zwei Unterknoten → knoten durch nächst kleineren/grösseren ersetzen und löschen

Laufzeit

Worst Case: immer $O(Tiefe)$

↳ Tiefe, wenn balanciert = $\log N$

⇒ Average: immer $O(\log N)$

Eigenschaften

Effizient, balanciert, sortiert

2-3 Baum

→ Art von balanciertem Baum, bei dem jeder Knoten zwei oder drei Unterknoten haben kann

Funktionsweise

Suche

- gleich wie binär; Zusätzlich: wenn gesuchter Wert zwischen den Werten im 3er Knoten ist, dann nehme den mittleren Unterknoten

Einfügen

- Suchen nach Wert → wenn kein Unterknoten mehr existiert → einfügen
- 2er Knoten wird zu 3er Knoten
- 3er Knoten wird temporär zu 4er Knoten und der mittlere Wert wird einen Knoten nach oben geschoben

Löschen

- Suchen
- Löschen
- Baum wieder anordnen das 2-3 Regel erfüllt ist → 3 Umstrukturierungs Fälle

Laufzeit

- Alle Blätter haben die selbe Tiefe: $O(\overbrace{\log N}^{\text{Tiefe}})$ $\left(\overbrace{c}^{\text{konstante ändert mit Implement.}} \log N\right)$

Eigenschaften

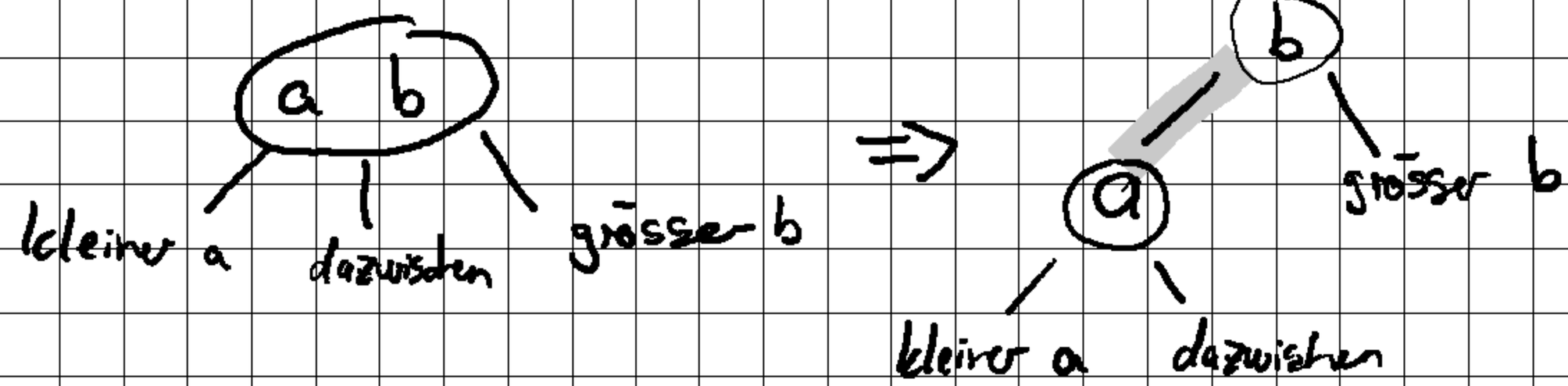
Effizient & Anpassungsfähig für grosse, sich ändernde Datenstrukturen

Rot - Schwarz Baum

→ implementierungs Methode für 2-3 Bäume

↳ left-leaning rot-schwarz Bäumen

⇒ wandle 2-3 Baum in binären Baum um, wobei ein 3er knoten wie folgt geteilt wird



Hier wird die markierte kante "rote kante" genannt und stellt somit einen virtuellen 3er knoten dar.

bei einem left-leaning baum, sind rote kanten nur immer linke kanten.

⇒ Jedes Blatt hat immer noch dieselbe Tiefe, man zählt die roten kanten nicht als Tiefe.

Ebenfalls darf ein knoten nicht mehr als 1 rote kante haben

B - Bäume

→ die generalisierte Form von 2-3 Bäumen

↳ ein Baum kann also auch grösser als 3 Kinder haben

Anwendungen

- SQL
- Linux kernel Scheduler
- Windows HPFS (High performance File System)

Hashing

→ Funktion um ein Wort (Schlüssel) in einen Zahlenwert zu konvertieren, der als Index in einem Array dient

⇒ generieren zwei Schlüssel den gleichen Hash-Wert führt dies zu einer Kollision

Ideale Hash-Funktion

→ Effizient & jeder Wert hat die gleiche Wahrscheinlichkeit erreicht zu werden.

↳ Dadurch werden die Schlüssel gleichmässig verteilt

⇒ Uniform Hashing Assumption

↳ M → Array-Grösse

↳ N → Anzahl Schlüssel

Aufbau Hash-Funktion

$(a \cdot k) \,\%\, M$ ~ Array Grösse

a → konstante

k → key

Seperate Chaining

→ Hash Verfahren um Kollisionen zu vermeiden

Funktionsweise

- Jeder Index besitzt eine Linked-List (prepend)
- Bei Kollision wird der Schlüssel am Anfang der Linked-List eingefügt
- Jeder Eintrag in die Linked-List enthält Schlüssel & Wert

Anzahl Elemente für jede Linked List

$\underline{N}$ ~ Anzahl Schlüssel

M ~ Array Grösse

$\Rightarrow$ M wird so definiert, dass der Array möglichst gefüllt ist

$$\Rightarrow M = \frac{N}{5}$$

Laufzeit

Worst case: $O(\log N)$

Average case: 3-5 (konstant)

Erweiterung

Two - Probe - Hashing

→ Für jeden Schlüssel werden mit zwei Hash Funktionen zwei Hash-Werte berechnet und der Schlüssel wird zu der kleineren Linked-List hinzugefügt.

Linear Probing

→ Hash verfahren, um kollisionen zu vermeiden

⇒ Tritt eine kolision auf, wird der Schlüssel im nächsten, freien Index abgelegt.

Wahl für Array Grösse

⇒ $n = 2N$ → für beste Such geschwindigkeit

Laufzeit

Worst Case : immer $O(\log N)$

Average : 3-5 (konstant)

Seperate Chaining vs Linear Probing

Separate Chaining

+ Einfacher delete zu implementieren
- Peformanz verschlechtert sich
+ weniger anfällig auf Clustering bei schlechten Hash Funktionen
- mehr Speicherplatz, da Linked-List Overhead erzeugen

Linear Probing

+ weniger Speicher Platz
+ Bessere Cache Performanz
- Clustering

Perfekter Hash

→ Hash-Funktion die mit möglichst kleinem M keine Kollisionen generiert

Double Hashing

→ bei einer Kollision wird der generierte Index mit einer zweiten Hash-Funktion erneut generiert

↳ Dies wird solange wiederholt, bis der Index frei ist.

Diese zweite Hash-Funktion könnte wie folgt aussehen:

$$index = (index + 1 + (a \cdot key) \,\%\, (M-1)) \,\%\, M;$$

a (3): Konstante; key (2): Schlüssel; M: Array Grösse

Cuckoo Hashing

→ Hashing des Schlüssels zu zwei Positionen

↳ einfügen an einer der beiden Positionen

↳ falls besetzt → verdrängten Schlüssel an seine Alternative einsetzen

↳ Dieser Schritt wird rekursiv ausgeführt.

Java Hashes

true = 1231

false = 1237

} Ergebnis der Methode x.hashCode();

Graphentheorie

Graph = G(V,E) V → Menge Knoten
E → Menge Kanten

Ungerichtete Graphen

⇒ max Kanten = $\frac{N \cdot (N-1)}{2}$ | min = $\frac{N+1}{2}$

→ Knoten die über Kanten miteinander verbunden sind.

Gerichtete Graphen (Digraphs)

→ Knoten die über Kanten verbunden sind, jedoch sind die Kanten nur "Einbahn-Strassen" (a → b)

Gewichtet gerichtete Graphen

→ gerichteter Graph, jedoch hat jede Kante einen "Wert" der "bezahlt" werden muss bei der Begehung der Kante

Begriffe der Graphentheorie

Pfad → Folge von Knoten, die über Kanten verbunden sind

Zyklus (Kreis) → Pfad bei dem der erste und letzte Knoten gleich sind.

Eule Tour → Zyklus, der jede Kante genau einmal verwendet

Hamilton Tour → Zyklus, der jeden Knoten genau einmal verwendet

verbunden → gibt es einen Pfad von einem Knoten zum anderen

Planarität → Graph Zeichnen ohne dass sich Kanten kreuzen

Graphimorphismus → Stellen zwei Adjazenzmatrizen den gleichen Graph dar

azyklisch → Graph ohne Zyklen

Adjazenzmatrizen

$A = (a_{ij}) \Rightarrow$ a_{ij} (von i zu j)

(gibt es keine explizite Kante von einem Knoten zu sich selbst dann $a_{ii} = 0$)

in Matrix: 1 wenn Kante existiert
0 wenn nicht

⇓

ist der Graph gewichtet schreiben wir Gewicht

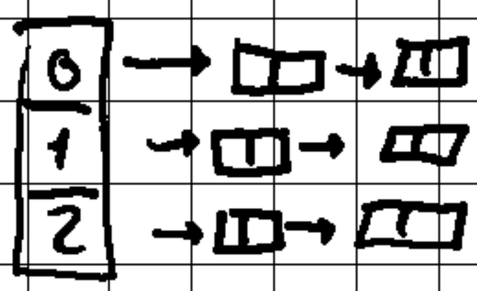
$A^n \Rightarrow$ jeder Eintrag zählt die Menge der Kantenzüge mit Länge n von i nach j

↳ $a_{ij}^2 = deg(a)$

Adjazenzliste

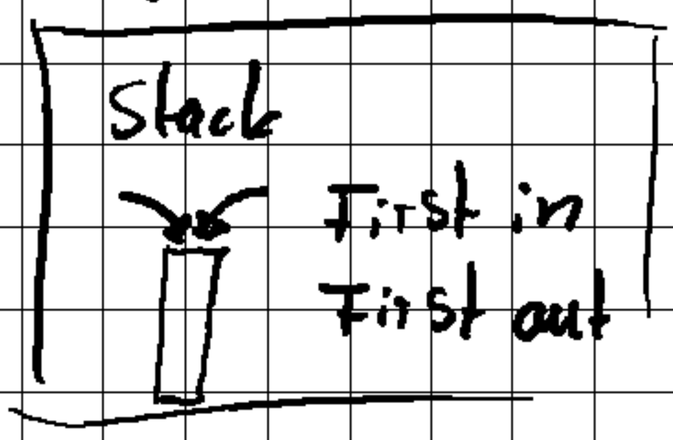
Array von Listen, indem die angrenzenden Knoten in einer Liste stehen.

Hat der Graph ein Gewicht, stehen in der Liste ebenfalls die Gewichte.



Tiefensuche (DFS)

→ Durchsuche einen Graphen, indem man alle angrenzenden Knoten, die noch nicht besucht wurden, auf einen Stack schreibt und das dann wiederholt für jedes Element im Stack, bis alle besucht wurden.

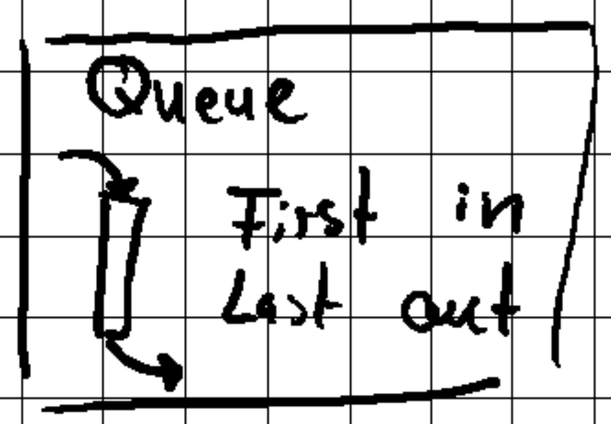


Laufzeit: $O(V+E)$

↳ Es werden alle Knoten & Kanten einmal besucht

Breiten Suche (BFS)

→ Durchsuche einen Graphen indem man alle angrenzenden Knoten, die noch nicht besucht wurden, auf eine Queue schreibt und dies dann für jeden Knoten in der Queue wiederholt, bis alle Knoten besucht wurden.



⇒ kann verwendet werden, um den kürzesten Pfad zwischen zwei Knoten zu finden.

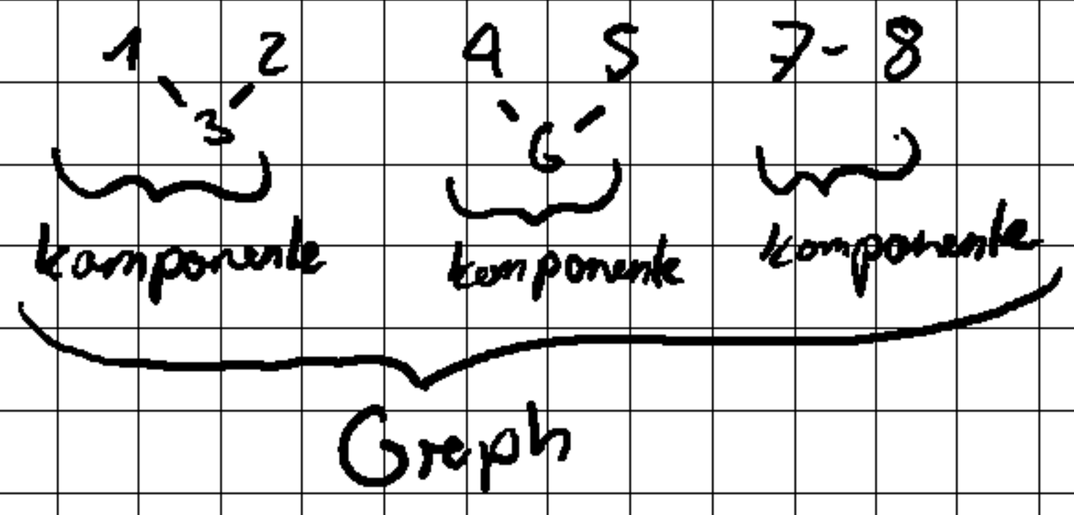
Laufzeit: $O(V+E)$

↳ Es werden alle Knoten & Kanten einmal besucht

Zusammenhangskomponente

→ maximale Menge an verbundenen Knoten

↳ Untergraph von einem anderen Graphen



Komponenten können mit DFS / BFS gefunden werden

⇒ Anzahl Durchläufe = Anzahl Komponenten

Starke Zusammenhangskomponente

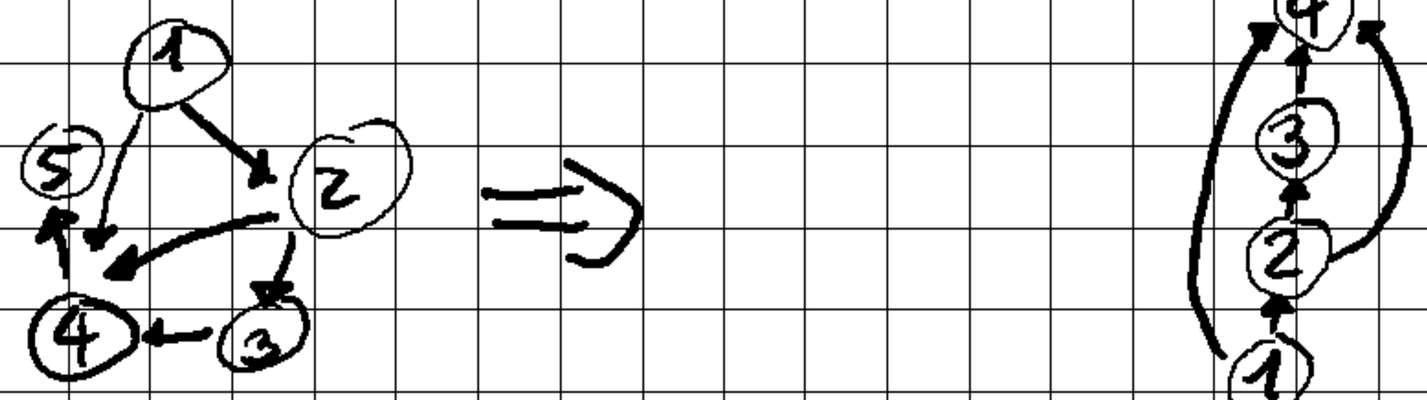
→ in einem gerichteten Graphen gibt es zwei Knoten die in beide Richtungen miteinander verbunden sind

↳ ist dies der Fall nennt man diese zwei Knoten stark zusammenhängend

gerichteter azyklischer Graph (DAG)

→ Graph umschreiben, sodass jede Kante nach oben zeigt

↳ topologisch sortieren



topologisches Sortieren

→ reverse Postorder

↳ Tiefensuche füge Element der Postorder an, wenn es keinen Unterknoten mehr gibt

Minimum Spanning Tree (MST)

→ ein aufspannender Baum eines Graphen, bei dem alle Gewichte der Kanten minimal sind im vergleich zu allen anderen möglichen aufspannenden Bäumen

⇒ Aufspannender Baum

↳ Verbindung aller Knoten ohne Zyklen (kreise)

↳ ist somit ein Subgraph

Existenz

Ein MST existiert, wenn ein Graph verbunden ist und alle Kantengewichte unterschiedlich sind

Schnitteigenschaft

→ Ein Schnitt teilt einen Graphen in zwei knoten Mengen

→ Eine Querkante verbindet die beiden Schnittmengen

Die Querkante mit dem kleinsten Gewicht muss Teil des MST sein

Greedy Algorithmen

→ Wiederholen der Schnitteigenschaft für alle möglichen Schnitte ergibt den MST

Kruksal Algorithmus

Funktionsweise

- Sortiere alle kanten nach ihrem Gewicht
- nimm die kleinste kante, überprüfe ob sie einen Zyklus mit dem bis jetzt aufgespannten Baum bildet und füge sie hinzu falls nicht
- wiederhole bis V-1 kanten im aufgespannten Baum sind.

Laufzeit

Sortieren der kanten : $O(E \log E)$

Union - Find : konstant $O(E \alpha)$

$\Rightarrow$ gesamt wird vereinfacht als $O(E \log E)$

Prim's Algorithmus

Funktionsweise

- Wähle einen Startknoten
- nehme die Kante mit dem kleinsten Gewicht, die einen Knoten im MST mit einem ausserhalb verbindet, ohne dass dabei ein Zyklus entsteht.
- wiederholen bis alle Knoten Teil des MST sind

⇒ Um die Kante mit dem kleinsten Gewicht zu finden, wird eine Priority Queue verwendet.

Laufzeit

→ hängt von der Implementation der Priority Queue ab.

⇒ Binär-Heap, 4-Way-Heap, Fibona Heap

Lazy Approach: $O(E \log E)$

→ alle Kanten ausserhalb des MST befinden sich in der Priority Queue und werden einfach ignoriert, nachdem ein Knoten dem MST hinzugefügt wurde

⇒ Lazy, weil immer noch zuerst überprüft werden muss, ob eine Kante bereits Teil des MST ist.

Eager Approach: $O(E + V \log V)$

→ Priority Queue wird stets aktuell gehalten

Kürzester Pfad

→ Pfad mit geringstem Gewicht zwischen zwei Knoten

Grundidee:

- Distanz von A ist 0
- Distanz von B ist der Weg von B nach A
  . Distanz
- falls B keine direkte Verbindung nach A hat → ∞
- Mache dies für alle Tiefen von Kantenzügen und finde so alle möglichen Züge von B nach A

Dijkstra's Algorithmus

Funktionsweise (funktioniert nur bei positiven Gewichten)

- setze alle bekannten Distanzen zum Startknoten auf ∞
  ausser den Startknoten der wird auf 0 gesetzt
- wähle den knoten mit der kleinsten Distanz, der noch nicht
  besucht wurde und nehme ihn als Start knoten
- Überprüfe jeden unbesuchten Nachbarn des ausgewählten knotns
  Aktualisiere die kürzeste Distanz zu diesem Nachbarknoten, wenn
- der Pfad über den ausgewählten knoten kleiner ist als die Bekannte
- markiere den ausgewählten knoten und besuche sie nicht erneut
- wiederhole bis der Ziel knoten erreicht wurde

Laufzeit

kommt auf die Implementierung der Priority Queue drauf an

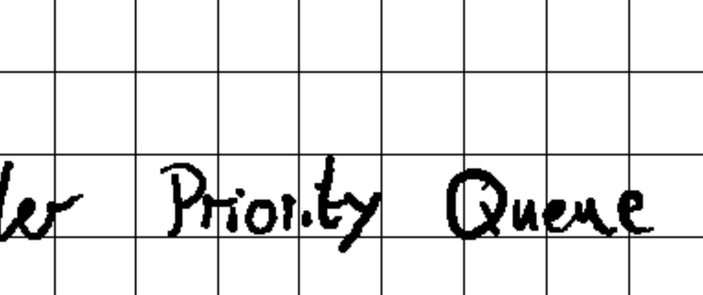
binär-Heap: $O((V+E)\log V)$

Fibonacci-Heap: $O(E+V\log V)$

Visualisierung

Q:	A	B	C	D	E
0	0	∞	∞	∞	∞
1		10	3	∞	∞
2		7		11	5
3		7		11	
4				9	

Konkanzüge ↓



S: {A, C, E, B, D}

S: Reihenfolge besuchter knoten

Q: kosten von A nach B-E

Bellman-Ford Algorithmus

→ Berechnung kürzester Pfad, auch wenn negative Gewichte vorkommen

Funktionsweise

- Setze alle Distanzen auf ∞ ausser die des Startknoten dieser wird auf 0 gesetzt
- Für jeden knoten im Graphen, gehe alle kanten durch und überprüfe ob der Pfad vom Startknoten zum aktuellen knoten kürzer ist, als die bereits bekannte. Aktualisie wenn ja
- Gehe erneut durch alle kanten.
  Falls der kürzeste Pfad aktualisiert werden kann enthält der Graph einen negativen Zyklus
  ↳ somit gibt es keinen kürzesten Pfad

⇒ Bellman-Ford kann somit den kürzesten Pfad nicht berechnen, jedoch erkennt er negative Zyklen

Laufzeit:

$O(V \cdot E)$