Algorithmen und Datenstrukturen

1. Allgemeines

Auf/Abrunden: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

$\lfloor 3,7 \rfloor = 3 \qquad \lceil 3,1 \rceil = 4 \qquad \lceil \frac{a}{b} \rceil \leq \frac{a+(b-1)}{b}$

Modulo $\boxed{a\%n = a \mod n = a - \left(\left\lfloor \frac{a}{n} \right\rfloor \cdot n \right)}$

Gesucht r mit $a = nq + r \qquad 0 \leq r < n, q \in \mathbb{Z}$

Fakultät: $n! = \begin{cases} n \cdot (n-1)! & \text{für } n \geq 1 \\ 1 & \text{für } n = 0 \end{cases}$

Fibonacci-Zahl: $f(n) = \begin{cases} f(n-1) + f(n-2) & \text{für } n \geq 2 \\ 1 & \text{für } n = 1 \\ 0 & \text{für } n = 0 \end{cases}$

Fibonacci-Folge: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

2. Algorithmen

Ein Algorithmus ist ein Verfahren mit einer **präzisen** (d.h. in einer genau festgelegten Sprache abgefassten) **endlichen** Beschreibung, unter Verwendung **effektiver** (tatsächlich ausführbarer), **elementarer** Verarbeitungsschritte. Ein Algorithmus besitzt eine oder mehrere Eingaben (Instanz mit Problemgröße n) und berechnet daraus eine oder mehrere Ausgaben.
Die Qualität eines Algorithmus ergibt sich aus seiner Effizienz, Komplexität, Robustheit und Korrektheit.

2.1. Darstellungsarten

- Flussdiagramm (Statements als Blöcke, verzweigte Bedingungen)
- Struktogramm (Pseudocode, Bei Bedingung mehrere Spalten)
- Programmiersprache/Pseudocode

2.2. Elementare Bausteine

- Elementarer Verarbeitungsschritt (z.B. Zuweisung Variable)
- Sequenz
- Bedingter Verarbeitungsschritt (if/else)
- Wiederholung (for/while)

2.3. Eigenschaften

Determiniert: Der Algorithmus liefert bei gleichen Startbedingungen das gleiche Ergebnis.
Deterministisch: Die nächste anzuwendende Regel ist zu jedem Zeitpunkt definiert.

2.4. Algorithmenmuster (Design Patterns)

2.4.1 Divide and Conquer

Rekursive Rückführung eines zu lösenden Problems auf mehrere identische Probleme mit kleinerer Eingabemenge bis zum Trivialfall
Beispiele: Binäre Suche, MergeSort, Quicksort

2.4.2 Greedy

Schrittweise Erweiterung der Lösung ausgehend von Startlösung unter Berücksichtigung des bestmöglichsten Schrittes
Beispiele: Berechnung Wechselgeld, Glasfasernetz

2.4.3 Brute Force

Erzeuge alle in Frage kommenden Kandidaten und suche besten aus

2.4.4 Backtracing

Systematische Suchtechnik, um Lösungsraum vollständig abzuarbeiten
Beispiele: Labyrinth (mit frühem Abbrechen von falschen Lösungspfaden)

2.4.5 Dynamisches Programmieren

Statt Rekursion berechnet man vom kleinsten Teilproblem "aufwärts". Zwischenergebnisse werden in Tabellen gespeichert
Beispiele: Fibonacci

2.5. Effizienz

Die Effizienz eines Algorithmus ist seine Sparsamkeit bezüglich der Ressourcen, Zeit und Speicherplatz, die er zur Lösung eines festgelegten Problems beansprucht.

2.6. Komplexität

Schrankenfunktionen: $1 < \log_{10}(n) < \ln(n) < \log_2(n) < \sqrt{n} < n < n \cdot \ln(n) < (\log n)! < n^2 < e^n < n! < n^n < 2^{2^n}$
Aber $\{\log_{10}(n), \ln(n), log_2(n)\} \in \Theta(\log n)$
Landau-Symbole:

Notation	Definition	
$f \in \mathcal{O}(g(n))$	$0 \leq f(n) \leq c \cdot g(n)$	$\forall n > n_0$
$f \in \Omega(g(n))$	$f(n) \geq c \cdot g(n) \geq 0$	$\forall n > n_0$
$f \in \Theta(g(n))$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$\forall n > n_0$

2.6.1 Asymptotische Notation

Notation	Grenzwertdef. (falls existent)	Wachstum
$g \in \mathcal{O}(f)$	$0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$	nicht schneller als f
$g \in \Omega(f)$	$0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$	nicht langsamer als f
$g \in \Theta(f)$	$0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$	gleich wie f
$g \in o(f)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$	langsamer als f
$g \in \omega(f)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$	schneller als f

2.6.2 Rechenregeln

$c \cdot f(n) = \mathcal{O}(f(n))$ für eine beliebige Konst. c
$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n)g(n))$
$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(max\{f(n), g(n)\})$
Gilt auch für Ω und Θ.

2.6.3 Typische Laufzeitklassen

	Laufzeittyp	Beispiel
$\Theta(1)$	konstant	Löschen von erstem El. aus Liste
$\Theta(\log n)$	logarithmisch	Suchen in sortierter Liste
$\Theta(n)$	linear	Suchen in unsortierter Liste
$\Theta(n \log n)$	loglinear	Quicksort
$\Theta(n^2)$	quadratisch	Insertion Sort
$\Theta(n^3)$	kubisch	Matrix-Matrix-Multiplikation
$\Theta(2^n)$	exponentiell	Traveling Salesman

Beispiele:
- Array: elementAt $\mathcal{O}(1)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$
- LinkedList: elementAt $\mathcal{O}(n)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$
- Stack (Array/LinkedList): push, pop, top $\mathcal{O}(1)$
- Queue (LinkedList): push, pop, top $\mathcal{O}(1)$

2.6.4 Rekurrenzen

Substitutionsmethode: Lösung raten, einsetzen und mit Induktion beweisen.

Mastertheorem:

Gegeben: $\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \text{ mit } a \geq 1, b > 1}$

$a \geq 1$: Anzahl der Unterprobleme innerhalb einer Rekursionstiefe (meist 1 oder 2)
$b > 1$: Faktor, um den jedes Unterproblem verkleinert ist.
$\epsilon > 0$: Eine beliebige Konstante.
$f(n)$: Aufwand der durch Division des Problems und Kombination der Teillösungen entsteht (nicht rekursiver Anteil, von $T(n)$ unabhängig).

1. Fall: $f(n) \in \mathcal{O}\left(n^{\log_b(a)-\varepsilon}\right)$
   Dann ist $T(n) \in \Theta\left(n^{\log_b a}\right)$
2. Fall: $f(n) \in \Theta\left(n^{\log_b a} \log(n)^k\right)$
   Dann ist $T(n) \in \Theta\left(n^{\log_b a} \log(n)^{k+1}\right)$
3. Fall: $f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$
   Dann ist $T(n) \in \Theta(f(n))$

2.7. Robustheit

Die Robustheit eines Algorithmus beschreibt die Fähigkeit auch bei ungünstigen Eingaben korrekt und effizient zu terminieren.

2.8. Korrektheit

Ein Algorithmus heißt korrekt, wenn er für jede Eingabeinstanz mit korrekter Ausgabe terminiert.
Qualitätskontrolle:

- Überprüfung mit geeigneten Eingabedaten die alle möglichen Fälle testen. Deckt einzelne Fehler auf, aber Fehlerfreiheit nicht garantiert.
- Formaler Beweis mit Hoare-Kalkül etc. meist sehr schwierig.

2.9. Rekursion

Eine Funktion ruft sich selbst auf bis ein Abbruchereignis eintritt. Danach werden die Rückgabewerte in umgekehrter Reihenfolge verkettet.
Beispiel Fakultät: $\text{fac}(n) = n \cdot \text{fac}(n-1), \text{fac}(1) = 1$

3. Datenstrukturen

Eine Datenstruktur ist eine **logische Anordnung** von Daten mit **Zugriffs- und Verwaltungsmöglichkeiten** der **repräsentierten Informationen** über Operationen.
Eine Datenstruktur besitzt:
- Menge von Werten
- Literale zum Bezeichnen von Werten
- Menge von Operationen auf die Werte

3.1. Stapel (Stack)

Basieren auf dem LIFO (last in first out) Prinzip.

`push(a)`	Legt ein neues Element a oben auf den Stack
`pop()`	Entfernt das oberste Element vom Stack
`top()`	Gibt das oberste Element vom Stack zurück.

Implementierung: entweder als Array (`top()` ist Zeiger auf letztes Element oder als Linked List (Jedes Element zeigt auf das Element darunter)

3.2. Warteschlange (Queue)

`*front`	zeigt auf das erste Element der Warteschlage
`*back`	zeigt auf das Ende der Warteschlange.
`enqueue(x)`	x am Ende der Warteschlange hinzufügen.
`dequeue()`	Erstes Element aus der Warteschlange nehmen und zurückgeben.

Implementierung z.B. als abgewandelte Linked List, d.h. Zeiger auf erstes und letztes Element werden gespeichert und jedes Element zeigt auf das nächste.
Priority Queue: Queue mit zugeordnetem Schlüssel, Entfernen von Element mit minimalem Schlüssel, Speicherung in fast vollständigem Binärbaum

3.3. Feld/Liste

`elementAt(i)`	Element in Position `i`
`insert(d, i)`	Element `d` an Position `i` einfügen
`erase(i)`	Element an Position `i` entfernen
`size()`	Gibt Größe/Länge des Feldes zurück

3.3.1 Vorteile und Nachteile der Implementierungen

DS	Vorteile	Nachteile
Array	direkter Zugriff mit $A[i]$ sequ. Durchlaufen einfach	Verlängern aufwendig Hinzufügen löschen aufwendig
Linked List	sequ. Durchlaufen einfach dynamische Länge Einfügen/Löschen einfach	Zugriff auf i-tes Element zusätzlicher Speicher für Ptr
Doubly Linked List	Durchlauf beide Richtungen Einfügen/Löschen einfacher	zusätzlicher Speicher für Ptr Referenzverw. aufwändig

3.4. Union Find

Verwaltet die Partitionierung einer Menge in disjunkte Teilmengen.

`makeset(x)`	Fügt Element hinzu
`find(x)`	Findet die Menge, die x enthält
`union(x, y)`	Die Funktion fügt die beiden Mengen zusammen

Implementierung: Als Linked List: Set = Liste. Jedes Element zeigt auf auf das nächste Element und zusätzlich auf das Set, in dem es enthalten ist. `find()` gibt einfach den Zeiger auf das enthaltende Set zurück. `union()` hängt die beiden Listen aneinander und überschreibt alle Zeiger auf eins der Sets. Auch Baumimplementierung ist möglich.

3.5. Augmentierung/Abänderung

1. Wähle eine zugrundeliegende Datenstruktur
2. Welche zusätzliche Information wird gebraucht?
3. Zeige: Diese Informationen können effizient geupdated werden wenn die Datenstruktur geändert wird
4. Entwickle die Operationen

4. Sortieralgorithmen

in-place: Nur konstanter Hilfsspeicher nötig. $S : \mathcal{O}(1)$
out-of-place: Zusätzlicher Speicher abhängig von n nötig. $S : \mathcal{O}(f(n))$

4.1. Insertion-Sort

1. Wähle beginnend bei 2 das nächste Element.
2. Solange es kleiner als seine Vorgänger ist, tausche es.

Im schlimmsten Fall $\frac{n}{2}(n-1)$

Best-case: $\mathcal{O}(n)$, Worst-case: $\mathcal{O}(n^2)$

```
INSERTIONSORT(A)
1 for i = 2 to Länge(A) do
2   key ← A[i]
3   j = i
4   while j > 1 and A[j-1] > key do
5     A[j] = A[j - 1]
6     j = j - 1
7   A[j] = key
```

4.2. Quicksort $\mathcal{O}(n^2)$, avg: $\mathcal{O}(n \log n)$

1. Wähle ein Pivotelement, welches die Liste in zwei Hälften teilt.
2. Sortiere die Liste so um, dass Elemente, die kleiner als das Pivotelement, in der einen Hälfte und größere in der anderen Hälfte sind. Suche dazu mit zwei Laufvariablen das Feld ab, bis jede eine unpassende Variable gefunden hat, dann tausche diese.
3. Wiederhole die Schritte 1. bis 3. mit beiden Teillisten, bis jede Teilliste sortiert ist.

```
QUICKSORT (A, p, r)
1 if p < r {
2   q = PARTITION (A, p, r)
3   QUICKSORT (A, p, q - 1)
4   QUICKSORT (A, q + 1, r)
5 }
```

```
PARTITION(A, p, r)
1 x = A[r] //Pivotelement
2 i = p-1
3 for j = p to r - 1 {
4   if A[j] ≤ x {
5     i = i+1
6     vertausche A[i] ↔ A[j]
7   }
8 }
9 vertausche A[i + 1] ↔ A[r]
10 return i + 1
```

4.3. Mergesort $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) \Rightarrow \mathcal{O}(n \log n)$

Feld in der Mitte rekursiv halbieren, bis Feldlänge $= 1$
Teilsortierte Felder zusammenfügen (Reißverschluss)

```
mergeSort(List* a)
1. if (a→length() <= 1)
2.   return a;
3. List* b = a→half();
4. a = mergeSort(a);
5. b = mergeSort(b);
6. return merge(a,b);

half()
1. Node* ptr = head;
2. for (0 bis length/2-2)
3.   ptr = ptr→next;
4. List* L = new List(ptr→next);
5. ptr→next = NULL;
6. return L;
```

```
elAt() entspricht elementAt()
merge(a, b)
1. if (a→empty())
2.   delete a; return b;
3. if (b→empty())
4.   delete b; return a;
5. List* h;
6. if a→elAt(0) < b→elAt(0)
7.   h = a; else h = b;
8. int d = h→removeFront();
9. List L = merge(a,b);
10. L→insertFront(d);
11. return L;
```

4.4. Laufzeiten und Speicherbedarf von Sortieralgorithmen

Die Laufzeit bzw. Taktzyklen in Abhängigkeit einer (meist großen) Eingabemenge n.

Name	Best	Avg	Worst	Zusätzlicher Speicher
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	in-place
Merge	$\mathcal{O}(n \cdot \log_2 n)$	←	←	$\mathcal{O}(n)$
Heap-Sort	$\mathcal{O}(n \cdot \log_2 n)$	←	←	in-place
Quick	$\mathcal{O}(n \cdot \log_2 n)$	←	$\mathcal{O}(n^2)$	in-place

in-place bedeutet zusätzlicher Speicher von $\mathcal{O}(1) = \text{const.}$
Vergleichende Sortieralgorithmen brauchen mindestens $\Omega(n \log_2 n)$ Vergleiche, egal wie clever sie sind!

4.5. Suchalgorithmen

sequentielles Suchen: Feld durchlaufen und vergleichen
Laufzeit: $\mathcal{O}(n)$, geeignet für statische kleine Mengen
binäres Suchen: Vorsortieren, Vergleich mit Mitte
Laufzeit: $\mathcal{O}(\log n)$, geeignet für statische große Mengen
binärer Suchbaum: für dynamische Mengen

5. Graphen

$G = (E, V)$, mit E: Kanten, V: Knoten
Adjazenzmatrix $(V \times V)$: 1 oder 0 für Verbindung oder Gewicht.
Bei ungerichtetem Graph symmetrisch, sinnvoll wenn Graph fast vollständig
Adjazenzliste: Für jeden Knoten alle Nachbarknoten angeben mit Startknoten s

Ungerichteter Graph: Richtung der Kante spielt keine Rolle
Gerichteter Graph: Richtung der Kante spielt eine Rolle, Schleifen möglich (u, u) mit $u \in V$
DAG: Directed acyclic graph.

5.1. Eigenschaften

v **adjazent** zu u: $(u, v) \in E$ bzw. $\{u, v\} \in E$ für $u, v \in V$
Inzident: Knoten v und Kante $e = (x, v)$ bzw. $e = \{x, v\}$ mit $v, x \in V$

Pfad: Folge von miteinander verbundenen Knoten
einfacher Pfad: alle Knoten sind paarweise verschieden
Zyklus: Pfad mit gleichem Anfangs- sowie Endknoten
Kreis: einfacher Pfad mit gleichem Anfangs- sowie Endknoten

5.1.1 Gerichteter Graph

Anzahl der eintretenden Kanten in v heißt Eingangsgrad: $indeg(v)$
Anzahl der austretenden Kanten aus v heißt Ausgangsgrad: $outdeg(v)$

stark zusammenhängend: jeder Knoten ist von jedem anderen Knoten erreichbar

starke Zusammenhangskomponente: stark zusammenhängender Teilgraph

Gibt alle zusammenhängenden Komponenten eines gerichteten Graphen aus

```
dfsTarjan()
1. dfsCount = 1;
2. foreach v ∈ V
3.   v->state = initial;
4.   v->parent = NULL;
5. foreach s ∈ V
6.   if (s->state == initial)
7.     dfsVisitTarjan(s)
```

```
dfsVisitTarjan(v)
1. v->state = active;
2. v->num = dfsCount++;
3. $.push(v);
4. foreach (x ∈ N[v])
5.   if (x->state == initial)
6.     x->parent = v
7.     dfsVisitTarjan(x);
8. if (cut parent edge of v)
9.   repeat
10.    x = $.pop();
11.    // add x to current component
12.  until (v == x)
13.  // print current component
```

5.1.2 Ungerichteter Graph

Anzahl der Kanten an v heißt Grad: $deg(v)$

zusammenhängend: jeder Knoten ist von jedem anderen Knoten erreichbar

5.2. Minimaler Spannbaum

Kruskal-Algorithmus $(\mathcal{O}(m \log n))$:
1. Sortiere alle Kanten aufsteigend nach Gewicht.
2. Wähle immer die nächst schwerere Kante, falls sie keine Schleife bildet.

```
PrimMST(s)
1. S = new PriorityQueue()
2. foreach v ∈ V
3.   v→key = ∞
4.   v→par = NULL;
5.   v→han = S→insert(v);
6. s→key = 0;
7. S→decreaseKey(s→han, 0);
8. while(!S→empty())
9.   v = S→extractMin();
10.  foreach x ∈ N[v]
11.    if (x→key > w(v,x))
12.      S→decreaseKey(x→han,
.          w(v,x));
13.      x→key = w(v,x);
14.      x→par = v;
```

5.3. Kürzeste Wege

SSSP (single source shortest path)
APSP (all pairs shortest path)

Satz: Teilpfade von kürzesten Pfaden sind auch kürzeste Pfade!

Dijkstra-Algorithmus: nur positive Kantengewichte!
Menge S: Knoten mit dem Gewicht ihres kürzesten Pfades von s
Menge Q: Min-Prioritätswarteschlange mit den ungeprüften Knoten.
Relaxationsschritt: Überprüfe ob ein Umweg über einen anderen Knoten kürzer ist.
Muss für jede Kante nur genau einmal überprüft werden.

```
BELLMANFORD(s)
Laufzeit: O(m · n)
1. foreach v ∈ V
2.   v→key = ∞;
3.   v→par = NULL;
4. s→key = 0;
5. for i = 1 to n-1
6.   foreach (x,y) ∈ E
7.     if (y→key > x→key + w(x,y))
8.       y→key = x→key + w(x,y);
9.       y→par = x;
10. foreach (x,y) ∈ E
11.   if (y→key > x→key + w(x,y))
12.     y→par = x;
```

```
APSP - Floyd Warshall
Laufzeit: O(m · n^2)
1. foreach (u,v) ∈ V
2.   d[u,v] = ∞
3.   pred[u,v] = NULL;
4. foreach v ∈ V
5.   d[u,v] = 0;
6. foreach (u,v) ∈ E
7.   d[u,v] = w(u,v);
8.   pred[u,v] = u;
9. foreach v ∈ V
10.  for {u,w} ∈ V × V
11.    if (d[u,w] > d[u,v] + d[v,w])
12.      d(u,w) = d(u,v) + d(v,w);
13.      pred(u,w) = pred(v,w);
```

5.4. Bäume

... sind spezielle Graphen mit einer Wurzel, Zweigen und Blättern.
Er besitzt keine zyklischen Strukturen und ist zusammenhängend.

Begriffe:
Grad deg(v): Anzahl der Kinder (direkten Nachfolger)
Blatt: Knoten mit $deg(v) = 0$
Tiefe $d(v)$: Länge des Pfades von der Wurzel bis zum Knoten v.
Höhe $h(v)$: Längster Pfad von v zu einem Blatt.
Niveau: Knoten mit gleicher Tiefe.

5.5. Binärer Suchbaum (BSB)

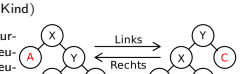
... ist ein geordneter Baum und $\forall v \in V : deg(v) \leq 2$
Ein Binärbaum ist vollständig, wenn $\forall v : deg(v) = 2 \ \vee \ deg(v) = 0$
Falls Höhe k, dann gilt: $2^{k+1} - 1$ Knoten und 2^k Blätter.

Bedeutung: Verdopplung der Eingabegröße, mit logarithmischer Vergrößerung der Struktur.
Binärbaum als verkettete Liste: Knoten x mit $links[x], rechts[x], parent[x], key[x]$

Balancierter Suchbaum: Höhe des Baumes ist immer in $\mathcal{O}(\log n)$

5.5.1 AVL-Baum

Betrag der Differenz der Höhe der Teilbäume für jeden Knoten ist ≤ 1.
AVL-Baum der Höhe h enthält mindestens $F_{h+2} - 1$ und höchstens $2^h - 1$ Knoten. Blattknoten sind Dummys ohne Daten. Jeder interne Knoten hat genau zwei Kinder Jeder Knoten speichert die Balance $v = h(\text{linkes Kind}) - h(\text{rechtes Kind})$
Rotationen:
Nach links: Rechtes Kind wird Wurzel, Wurzel wird linkes Kind der neuen Wurzel, Altes linkes Kind der neuen Wurzel wird rechtes Kind der alten Wurzel.
Nach rechts: Umkehrung der Rotation nach links



Einfügen:
1. Knoten einfügen wie in normalen BSB
2. Vom Elternknoten des eingefügten Knotens aufsteigend die Balance aller Knoten bis zur Wurzel prüfen und bei Bedarf die AVL-Bedingung wiederherstellen
Löschen:
1. Knoten löschen wie im normalen BSB
2. Vom Elternknoten des überbrücken Knotens aufsteigend die Balance aller Knoten bis zur Wurzel prüfen und bei Bedarf die AVL-Bedingung wiederherstellen
Balance eines Knotens wiederherstellen:

```
1 Balance(v)
2 if (v->balance == -2)
3   if (v->right->balance ∈ {0, −1})
4     v = LeftRotate(v)
5   else
6     v->right = RightRotate(v->right)
7     v = LeftRotate(v) // Doppelrotation
8 else
9   if (v->left->balance ∈ {0, 1})
10    v = RightRotate(v)
11  else
12    v->left = LeftRotate(v->left)
13    v = RightRotate(v) // Doppelrotation
14 return v
```

5.5.2 Traversierung

Pre-order: Wurzel → linker Teilbaum → rechter Teilbaum
In-order: linker Teilbaum → Wurzel → rechter Teilbaum
Post-order: linker Teilbaum → rechter Teilbaum → Wurzel

5.6. Suchen in Graphen

Breitensuche(BFS) $\Theta(|V| + |E|)$
Von einem Startknoten S werden alle noch nicht durchsuchten Knoten mit Abstand k durchsucht. Der Suchradius breitet sich aus. Neue Knoten kommen in eine Queue an zu besuchenden Knoten.
Code ähnlich wie DFS, nur Stack durch Queue ersetzt.

Tiefensuche(DFS) $\Theta(|V| + |E|)$
Von einem Knoten werden alle Nachfolger rekursiv durchsucht. Mehrere Verzweigungsmöglichkeiten werden zwischengespeichert.
1. DFS(G): Suche im Graph G den nächsten unbesuchten Knoten a. Rufe DFS-VISIT(a) auf.
2. DFS-VISIT(a): Finde rekursiv alle Nachfolgerknoten von a und markiere sie als durchsucht.

5.7. Heap

Fast vollständiger Binärbaum mit Indizierung von links nach rechts und von oben nach unten. Wert eines Knotens ist immer kleiner bzw. größer als Werte der Kinder. Max-Heap: Wurzel hat den größten Wert, Min-Heap: Wurzel hat den kleinsten Wert.
Heap-Sort:
1. Erzeuge Max-Heap aus A
2. Wähle $|A|/2$ als Starknoten, da größter Knoten mit $deg > 1$
3. Betrachte Knoten i und seine beiden Kinder $2i$ und $2i + 1$

6. Hashtabellen

... sind Felder bei denen die Position eines Objekts durch eine Hashfunktion berechnet wird. Da es zu Kollisionen kommen kann, werden in den Feldern nur Verweise auf eine Liste gespeichert.
Schlüssel: wird von einem Schlüsselgenerator aus den Daten generiert.

6.1. Hashfunktion

... ordnet jedem Schlüssel aus einer großen Schlüsselmenge einen möglichst eindeutigen Wert aus einer kleineren Indexmenge zu. $h : key \rightarrow index$
Operatoren: Verkettete Hashtabelle: Jedes Feld entspricht einer Liste die mehrere kollidierte Daten speichern kann.
chained-hash-insert(T,x) : Füge x an den Kopf der Liste $T[h(x.schluessel)]$
chained-hash-search(T,k) : Suche Element k in der Liste $T[h(k)]$
chained-hash-delete(T,x) : entferne x aus der Liste $T[h(x.schluessel)]$

7. Fouriertransformation

Transformation eines periodischen Signals aus dem Zeit in den Frequenzbereich → Darstellung als Überlagerung von Sinus- und Kosinusfunktionen
Theorem:
Jede periodische Funktion $x(t)$ mit Periode T lässt sich als $\sum_{i=-\infty}^{\infty} \alpha_i e^{j \frac{2\pi}{T} t}$ schreiben.

7.1. Diskrete Fouriertransformation (DFT)

Fouriertransformation ist Integration über alle komplexen Exponentialfunktionen ⇒ Näherung über Summe entlang des Einheitskreises (mit $n-ten$ Einheitswurzeln)
Geg.: $x_0, ..., x_{N-1}$ mit $x_k = x(\frac{T}{N}k)$. Berechne

$$\hat{x}_l = \frac{1}{N} \sum_{k=0}^{N-1} x_k e^{-j \frac{2\pi}{N} lk} \quad \forall l \in \{0, ..., N-1\}$$

⇒ DFT ist Polynomauswertung, einfache Implementierung $\mathcal{O}(n^2)$

7.2. Fast Fourier Transform (FFT)

Schnelle Polynomauswertung mit $N = 2^k$.
Input: Polynom P vom Grad N-1, Koeffizenten C
Output: $R[l]$ enthält $P[W_N^l]$

```
FFT(C, N, W_N)
1    if (N == 1) return C
2    C_odd = [C[1], C[3], ..., C[N − 1]]
3    C_even = [C[0], C[2], ..., C[N − 2]]
4    R_odd = FFT(C_odd, N/2, W_N^2)
5    R_even = FFT(C_even, N/2, W_N^2)
6    W = 1; // = W_N^0
7    for l = 0 to N/2 − 1
8      R[l] = R_even[l] + W * R_odd[l]
9      R[l + N/2] = R_even[l] − W * R_odd[l]
10     W *= W_N; // = W_N^l
11   return R;
```

Für DFT: $W_N = e^{-j \frac{2\pi}{N}}$ und Ergebnis durch N dividieren; $\mathcal{O}(n \log n)$

Asymptotische Notation

oder als Grenzwert betrachtet

nicht schneller als $f(n)$	$O(f(n)) = \{g(n) \mid \exists c>0\, \exists n_0>0\, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$ Oberschranke	$0 \leq \dots < \infty$	$g(n) \in O(f(n))$	nicht schneller
nicht langsamer als $f(n)$	$\Omega(f(n)) = \{g(n) \mid \exists c>0\, \exists n_0>0\, \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$ Unterschranke	≥ 0	$g(n) \in \Omega(f(n))$	nicht langsamer
gleich wie $f(n)$	$\Theta(f(n)) = \{g(n) \mid \forall C>0\, \exists n_0>0\, \forall n \geq n_0 : g(n) = c \cdot f(n)\} = O(f(n)) \cap \Omega(f(n))$	$0 < c < \infty$	$g(n) \in \Theta(f(n))$	$g(n)$ gleich wie $f(n)$
echt langsamer als $f(n)$	$o(f(n)) = \{g(n) \mid \forall C>0\, \exists n_0>0\, \forall n \geq n_0 : g(n) \leq C \cdot f(n)\}$	$=0$	$g(n) \in o(f(n))$	echt langsamer
echt schneller als $f(n)$	$\omega(f(n)) = \{g(n) \mid \forall C>0\, \exists n_0>0\, \forall n \geq n_0 : g(n) \geq C \cdot f(n)\}$	$=\infty$	$g(n) \in \omega(f(n))$	echt schneller

$\lim_{n\to\infty} \frac{g(n)}{f(n)} =$

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$

Erwartungszeit = $\sum_{i \in I}$ 相应Wahrscheinlichkeit · 实际Laufzeit

Rechenregel für O-Notation (oder auch Ω-Notation)

- $c \cdot f(n) \in \Theta(f(n))$ für jede Konstante $c > 0$
- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- $O(f(n) + g(n)) = O(f(n))$ falls $g(n) \in O(f(n))$

Master-Theorem

Sei a, b, c, d positive Konstante und $n = b^k$ mit $k \in \mathbb{N}_0$

d = # rekursive Aufrufe n = # inputs b = Teilproblem-Größe

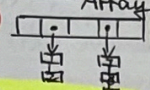
$a^b = n \Rightarrow b = \log_a n \quad \log_b b = 1$

$\log ab = \log a + \log b \quad \log \frac{a}{b} = \log a - \log b$

$\log a^b = b \log a \quad a^{\log_c b} = b^{\log_c a} \quad \log_b \sqrt{a} = (\frac{1}{b}) \cdot \log a$

$$r(n) = \begin{cases} a & \text{falls } n = 1 \\ cn + d \cdot r(\frac{n}{b}) & \text{falls } n > 1 \end{cases} \Rightarrow r(n) = \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log n) & \text{falls } d = b \\ \Theta(n^{\log_b d}) & \text{falls } d > b \end{cases}$$

Hashing mit Chaining (Verkettung)



⇒ Array mit Listen als Inhalt

Linear Sondieren / Probing

$h(k,i) = (h(k) + i) \bmod m$

+ kein zusätzliche Speicheraufwand − Löschen aufwendig

代入function. 若有Kollision ⇒ i++ (每次后移一格)

若末位已满. 从index 0 开始往后继续占用

删除element后. 为该Position排队的Elemente要向前补位

Perfektes Hashing

Element in Bucket = n # Kollision = $\frac{n \cdot (n-1)}{2}$

+ kein Kollision − nur statisch möglich (kein insert & remove) − hohe Speicherbedarf

Stufe 1: Elemente werden mit Funktion 1 auf unterschiedliche Buckets verteilt

确定最大冲突数 c $m = \lceil \sqrt{2cn} \rceil$ # insgesamt Kollisionen $\leq \sqrt{2cn}$

Stufe 2: 确定每个Bucket-Größe $m_e = c \cdot b_e \cdot (b_e - 1) + 1$ # Elemente im Bucket = b_e

然后为每个Bucket根据不同的 m_e 选择合适的Funktion 2 (那个不产生Kollision的Option)

Double Hashing

$h(k,i) = \lfloor h(k) + i \cdot h'(k) \rfloor \bmod m$ ($h'(k)$ ist teilerfremd zu m)

i = # Kollision i 初始都为0 有Kollision后自己Position的 i++ 直到找到无Kollision的Position停止

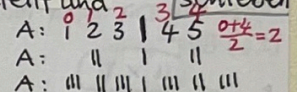
→ modifiziertes Double Hashing: 用Platzhalter标记gelöschte Element 来占位

若之后有新的Elemente要用这个Position ⇒ 新的Element 直接覆盖 Platzhalter

+ schnell finden der gehashten Elemente − Löschen mit Marken möglich

Sortierverfahren

- **Insertion (aufsteigend)** nächste Element ins links geordneten Teil an richtig Stelle einordnen
- **BubbleSort (aufsteigend)** jedes mal 2 Elemente betrachten aktuell > nächst ⇒ Position tausch
- **SelectionSort (aufsteigend)** jedes mal unsortierten Teil (verbleibenden Eingabesequenz) durchlaufen und das kleinste Element finden und es an Ausgabesequenz-Ende verschieben
- **MergeSort (aufsteigend)** Array rekursive in zwei Teilen aufgeteilt und Teilarray wird sortiert und zur Ursprungsarray mergen



- **Quicksort (aufsteigend)** Pivot auswählen und sortieren ≤ Pivot 和 > Pivot und dann rekursive in jeder Teilmenge Pivot wählen und sortieren
- **RadixSort (aufsteigend)** 将所有Elemente统一为相同长度(可补0) Sortiere zuerst entsprechend 最低位(个位) Ziffer und dann nacheinander für immer 更高位(十位、百位、千位) Stellen (为该Stelle排序时, 其余低位的顺序从上一次排序中保留下来)

数字优先于字母

	WorstC	AverageC	BestC	stabil
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✗
BubbleSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	✓
InsertionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	✓
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✗
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	✓
RadixSort	$O(\sim \cdot \ell) = O(n)$			✓

Ziffer-Länge

Binärbaum

jede Knote maximal 2 Kinder (Höhe h = max Tiefe t + 1)

Vollständiger Binärbaum

jede innere Knote genau 2 Kinder + alle Blätter dieselbe Tiefe t

$2^h - 1$ Knoten ($= 2^{h-1} - 1$ innere Knoten + 2^{h-1} Blätter)

Binär Heap G

1) G ist fast vollständig (Form-Invariante)

⇒ alle Ebene sind vollständig, außer der untersten Ebene

2) G erfüllt die Heap-Invariant

→ Min-Heap-Eigenschaft: key(v.parent) ≤ key(v)

→ Max-Heap-Eigenschaft: key(v.parent) ≥ key(v)

Node Index i: left child $2i$, right child $2i+1$, parent $\lfloor \frac{i}{2} \rfloor$

Priority Queue mit Binären Min-Heap

siftdown: zaul rekursiv mit kleinsten Nachfolger tauschen, wenn dieser kleiner ist

siftUp: zaul rekursiv mit Vorgänger tauschen, wenn dieser größer ist

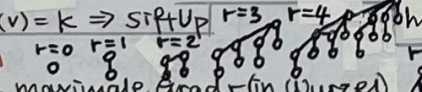
build: 根据给定顺序 ins Heap einfügen, 将前 $\lfloor \frac{n}{2} \rfloor$ 个Elemente 按与插入顺序相反的顺序 siftdown

insert: Element an unterste Ebene, linkeste freie Stelle einfügen und dann siftUp

deleteMin: Wurzel mit letzten Element tauschen und löschen, neue Wurzel siftdown

HeapSort: Wurzel wiederholt mit letzten Element tauschen + löschen, gelöschte Element hinten ans Array (absteigend sortiert)

decreaseKey: key(v) = k ⇒ siftUp



Binomial-Baum

maximale Tiefe r, maximale Grad r (in Wurzel)

insgesamte 2^r Knoten, in Tiefe $\ell \in \{0, \dots, r\}$ sind $\binom{r}{\ell} = \frac{r!}{\ell!(r-\ell)!}$ Knoten

Merge aus zwei Binomial-Bäume von Rang $r-1$

→ Wurzel mit dem kleinerem key wird neue Wurzel

Binomial Heap

besteht aus einer Menge Binomial-Bäume

→ jeder Binomial-Baum erfüllt die Min-Heap-Eigenschaft

→ für jeden Rang gibt es maximal einen Binomial-Baum

→ kleinste Element des Binär-Heaps ist eine der Wurzel (markiert durch Min-Zeiger)

→ ein Binomial-Heap mit n Knoten enthält maximal $1 + \lfloor \log_2 n \rfloor$ Binomial-Bäume

Min: liefert Wurzel mit Min-Zeiger zurück

merge: Bäume mit gleichen Rang mergen, andere aufnehmen

⇒ Binäraddition: $n_1 = 5_{10} = 101_2$ Knoten, $n_2 = 5_{10} = 101_2$ Knoten

$n' = \begin{array}{r} 101_2 \\ 101_2 \\ \hline 1010_2 \end{array} = 10_{10}$ Knoten

$2^3 = 8$ (r=3) $2^1 = 2$ (r=1)

insert: merge mit Binomial-Baum von Rang 0 (1 Element)

deleteMin: Wurzel (die mit Min-Zeiger) löschen → Baum zerfällt in Bäume ⇒ merge

decreaseKey: key(v) = k ⇒ siftUp ⇒ evtl. Min-Pointer aktualisieren

remove(v): decreaseKey(v, −∞), danach deleteMin

wenn imbalance passiert: ① Rotation direkt (nicht bis alle insert/remove Operation alles fertig) auf die (bottow to top) erste imbalanced Knote anwenden; ② immer nur drei Knoten für Rotation betrachten

Priority Queue - Laufzeit Tabelle

mittels	unsortierte Liste	sortierte Liste	binär Heap	binomial Heap	Fibonacci Heap
build	$O(n)$	$O(n \log n)$	$O(n)$		
insert	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$ (WorstC)
Min	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$ (WorstC)
deleteMin	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$ (amotisiert)
decreaseKey			$O(\log n)$	$O(\log n)$	$O(1)$ (amotisiert)
merge			$O(n)$	$O(\log n)$	$O(1)$ (WorstC)

einfachste | auch einfach, aber nicht so effizient | in place wenn minimal Speicher, Overhead wichtig | ein bisschen effizienter als Binär Heap | Implementierung Schwer

$\lfloor \frac{5}{2} \rfloor = \lfloor 2.5 \rfloor = 2$ 向下取整

$\lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil = 3$ 向上取整

Binär-Suchbaum

Schlüssel-Invariante: jede Knote hat einen eindeutigen key

Grad-Invariante: jede Knote hat maximal 2 Nachfolger

Suchbaum-Invariante: ∀ Knoten v:

⇒ linkTeil v.left: key(x) ≤ key(v)

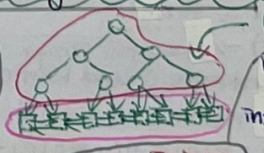
⇒ rechtTeil v.right: key(y) ≥ key(v)

intern: Nutzdaten an inneren Knoten speichern

extern: Nutzdaten in Blätter speichern

(mittels Zeiger auf Elemente einer sortierten Liste)

→ Navigationsstruktur



locate: starte in Wurzel, gehe links oder rechts abhängig von key

insert(e): mit locate e' finden ⇒ key(e') > key(e)? in Liste davor einfügen: Fehler

remove(k): mit locate k bis Element e in Liste erreicht, falls key(e) = k

→ lösche e aus Liste

→ lösche Vaterknote v = e.parent

→ setze w mit key(w) = k den neuen Wert

falls key(e) ≠ k ⇒ Fehler key(w) = key(v)

AVL-Baum

Binär-Suchbaum + Höhendifferenz $\{-1, 0, 1\}$ (ℓ − r)

Rebalancierung: locate, insert, remove ⇒ $O(\log n)$

insert: normal insert in Binär-Suchbaum und dann Höhendifferenz betrachten + aktualisieren

remove: 1) falls v ein Blatt oder genau 1 Kind hat, lösche v und ersetze v durch sein Kind

2) falls v zwei Kinder hat, vertausche v mit dem nächstkleineren (rechteste Knote im linken Teilbaum) und lösche v dort

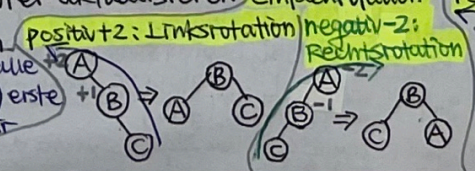
Doppelrotation

+2/−1: RL-Rotation −2/+1: LR-Rotation

Einfachrotation

positiv +2: Linksrotation

negativ −2: Rechtsrotation



(a,b)-Baum (externer Suchbaum)

Anzahl der Kinder

$a \geq 2$, $b \geq 2a-1$, Wurzel w: $2 \leq d(w) \leq b$

Tiefe $\leq 1 + \lfloor \log_a \frac{n+1}{2} \rfloor$ (für $n > 1$ Blätter)

locate	$\Theta(\log n)$
insert	$\Theta(\log n)$
remove	$\Theta(\log n)$

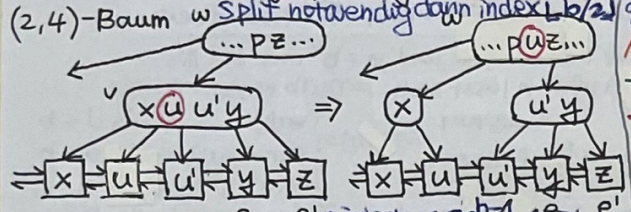
Form-Invariante: alle Blätter selbe Tiefe min/max $O(1)$

Grad-Invariante: für alle Knoten (außer Wurzel): $a \leq d(v) \leq b$

insert(e): key(e) in Knote v einfügen; $d(v) \leq b$ dann fertig; sonst v aufteilen, größte Key im linken Teil wird Vaterknote

(2,4)-Baum

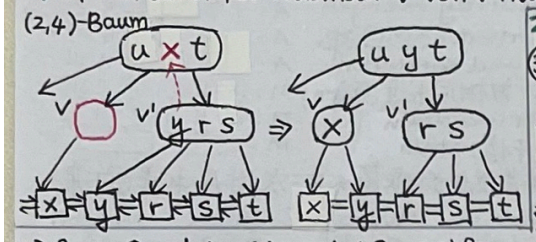
w Split notwendig dann index $\lfloor b/2 \rfloor$ geht in die Wurzel



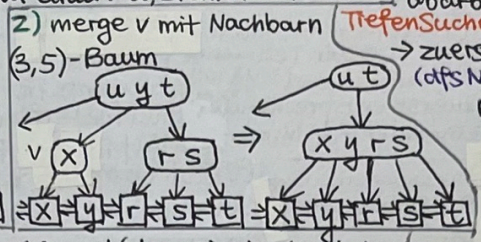
mindestens $2a^{h-1}-1$ Elemente

remove(K): k mit locate finden; in Knote v den Schlüssel (und Handel bzw. Element k in Knote v und Blatt k) von k entfernen; $d(v) \geq a$ dann fertig, sonst Kante stehlen

1) if ein direkter Nachbar v' von v hat Grad $> a$, stehle Kante von v'



2) merge v mit Nachbarn



→ falls Grad der Wurzel < 2: entferne Wurzel (das einzige Kind der alten Wurzel wird neue Wurzel)

Element in Wurzel löschen: ~~vertauschen mit nächst Kleinerem Element und dann weiter steal/merge~~

Zusammenhang von Graphen

- zusammenhängend ⇔ von jeden Knoten Pfad zu jedem anderen Knoten
- stark zusammenhängend ⇔ von jeden Knoten aus einem gerichteten Pfad zu jede anderen Knoten
- stark Zusammenhangskomponent ⇔ induzierter Teilgraph ist stark zusammenhängend und maximal

Topologische Sortierung in DAGs (directed acyclic graph 非循环)

für alle Kanten $e = (v,w)$ gilt topoNum(v) < topoNum(w)

1) wähle einen beliebigen Knote (hat keine eingehenden Kante bzw keinen Vorgänger) im DAG aus, füge ihn der Sortierung hinzu
2) Entferne den ausgewählten Knoten und alle von ihm ausgehenden Kante bzw. alle seine Vorgänger-Anzahl 1 abziehen
3) Wiederhole die Schritte 1 und 2 für die verbleibenden Knoten, bis DAG leer ist

kürzeste Weg finden: gemäß topologischer Sortierung die Distanz an Knote aktualisieren

Dijkstra-Algorithmus (Greedy-Strategie) $O(m+n \log n)$

→ nur für Graph mit nicht-negativ gewichtete Kanten

Ablauf: besuche die Knoten mit kleinster Gesamt-Distanz und Knotengewicht aktualisieren

→ Startknote s, Distanz zu jeder Knote auf ∞ gesetzt

→ Distanz zu allen Nachbarknoten gesetzt, Distanz nur neu gesetzt wenn neue Distanz kleiner als alte; wenn alle Knoten besetzt ⇒ fertig

Bellman-Ford-Algorithmus $O(m \cdot n)$ $m = |E|$ $n = |V|$

→ okay für Graph mit negativ gewichtete Kanten

durchlaufe $(n-1)$ mal alle Kanten im Graph und aktualisiere Distanz in jeder Iteration

→ wenn in n-te Iteration weitere Änderung gibt ⇒ es gibt Kreis negativer Länge

Floyd-Warshall-Algorithmus $O(n^3)$ $A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$

→ okay für Graph mit negativ gewichtete Kanten

für alle Knoten v überprüft, ob zwei Knoten u und v mit einem kürzeren Weg verknüpft werden können, der über v führt

Johnson-Algorithmus $O(m \cdot n + n^2 \log n)$ → okay für Graph mit negativ gewichtete Kanten

1) Hinzufügen der neuen künstlichen Startknote s und Kanten (s,v) für alle v mit $c(s,v) = 0$
2) Berechnen $d[s,v]$ mit Bellman-Ford-Algorithmus und setze Knotenpotenzial $\Phi[v] = d[s,v]$ für alle v
3) $\bar{c}(e)$-Werte für alle $e = (v,w)$ berechnen: $\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$
4) Distanz-Tabelle $\bar{d}$ mit modifizierten Kantengewichten via Dijkstra
5) korrekte Distanzen berechnen: $d[v,w] = \bar{d}[v,w] + \Phi[w] - \Phi[v]$ und dann Tabelle aktualisieren

Minimale-Spannbäume → Kanten mit minimalen Kosten nehmen, alle Knoten zu verbinden

Kruskal-Algorithmus $O(m \log m)$

Liste ungerichteter Kante $\{a,b\}, \{b,e\}, \{b,f\}, \{c,d\}, \{c,e\}, \{c,f\}, \{e,f\}$ ⇒ Graph wird partioniert ⇒ jede Knote hat eigene Teilgraph

Teilgraph iterativ über günstige Kante zusammengefügt (die noch nicht Teil des Spannbaums ist)

⇒ wähle wiederholt Kante mit minimalen Kosten (Kanten mit minimale Kosten, aber mit dem zu einem Kreis geführt werden kann, wird nicht gewählt)

⇒ Dies wird wiederholt, bis nur noch 1 Zusammenhangskomponent existiert

Prim-Algorithmus $O(m + n \log n)$

→ alle verbundene Kanten der bereits besuchten Knote betrachten und die Kante mit minimale Kantengewicht auswählen (aktuell minimal, nicht gesamt minimal)

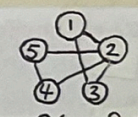
Kreis: alle Schleifen + alle "hin und rück" Kanten + alle normale Kreise mit mehr als 2 Knoten

```
for (int i=0; i<=n; i++)   O(n+1)
for (int i=1; i<=n; i++)   O(n)
```

$f_n = \frac{n+1}{\log n}$ $g_n = n$ $\lim_{n\to\infty} \left|\frac{f(x)}{g(x)}\right| = \lim \left|\frac{f(x)}{g'(x)}\right| = \frac{(1 \cdot \log n - \frac{1}{n} \cdot (n+1)) / (\log n)^2}{1} = \lim \frac{\log n - 1 - \frac{1}{n}}{(\log n)^2} = \lim \frac{1}{(\log n)^2} - \frac{1}{(\log n)^2} = 0$

$f_n = \sin(n)$ $g_n = 1$ $\lim_{n\to\infty} \left|\frac{f(x)}{g(x)}\right| = \lim \frac{\sin(n)}{1}$ = nicht definiert, aber $\lim_{n\to\infty} \sup(\sin(n)) = 1 \Rightarrow f(n) \in O(g_n)$

Graphenrepräsentation



Kantenliste: $\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{4,5\}$

\+ Speicherbedarf $O(m)$, insert $O(1)$, remove $O(1)$

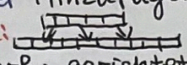
− find/remove $\Theta(m)$ (worst case), Nachbarn nur in $O(m)$ feststellbar

Adjazenzmatrix: $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$

\+ feststellen, ob zwei Knoten Nachbarn sind $O(1)$, insert/remove $O(1)$

− Speicherbedarf $\Theta(n^2)$, alle Nachbarn einer Knote finden $O(n)$, Hinzufügen neuer Knote ⇒ aufwendig (neue matrix)

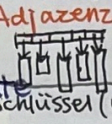
Adjazenzarray:



\+ Speicherbedarf: gerichtete Graph: $n+m+\Theta(1)$; ungerichtete Graph: $n+2m+\Theta(1)$

− insert/remove ⇒ aufwendig; deshalb nur für statische Graph geeignet

Adjazenzliste: + insert $O(1)$, remove $O(d)$ (per Handle in $O(1)$)



− Zeigerstruktur verbrauchen relativ viel Platz und Zugriffszeit

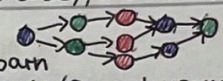
Adjazenzliste + Hashtabelle

→ Speicheraufwand: $O(n+m)$

→ Zeitaufwand: find/remove $O(1)$ (worst case), insert $O(1)$ (amortisiert)

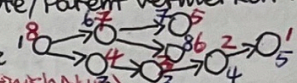
Rückwärts gilt für Kante (v,w)
{ dfsNum[v] > dfsNum[w]
{ finishNum[v] < finishNum[w]

BreitenSuche (BFS Breadth First Search)



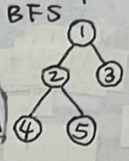
→ zuerst in die Breite gehen (alle Nachbarn abarbeiten), jeweils Distanz zu Startknote/Parent vermerken.

TiefenSuche (DFS Depth First Search)

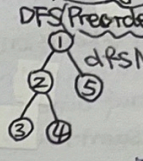


→ zuerst in die Tiefe gehen, Explorationsfolge (dfsNum) speichern, Fertigstellungsfolge (finishNum) speichern

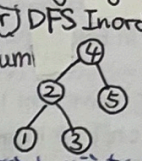
BFS



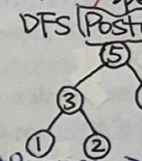
DFS (Preorder) dfsNum



DFS Inorder



DFS (Postorder) finishNum



Rückwärts entfernen ⇒ kreisfrei

Kürzeste-Weg-Problem

SSSP (single source shortest path): kürzeste Weg von einer Quelle zu allen anderen Knoten

APSP (all pairs shortest paths): kürzeste Weg zwischen allen Paaren

Hoffman Coding

message BCCABBDDAECCBBAEDDCC 原先 $20 \times 8 = 160$ bits

Character	count	Code
A	3	000
B	5	001
C	6	010
D	4	011
E	2	100

message $20 \times 3 = 60$ bits

\+ Tabelle $5 \times 8 + 5 \times 3 = 55$ bits

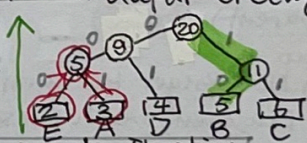
H insgesamt 115 bits



#inserts = 3 = #Schritte von T 顺时针去H

Hoffman-Baum wird von Blätter aufwärts erstellt

⇒ immer für die zwei Knoten mit aktuell minimale Gewicht (= count) einen Vaterknote mit deren Gewicht-Summe dafür erzeugen (Code: von Wurzel zu Blatt)



char	count	Code	Message
A	3	001	$3 \times 3 = 9$
B	5	10	$5 \times 2 = 10$
C	6	11	$6 \times 2 = 12$
D	4	01	$4 \times 2 = 8$
E	2	+ 000	$+ 2 \times 3 = 6$
		12 bits	45 bits

Tabelle $5 \times 8 + 12 = 52$ bits

\+ Message 45 bits

insgesamt 97 bits

#Gesamt Bits = #MessageBits + #Tabelle

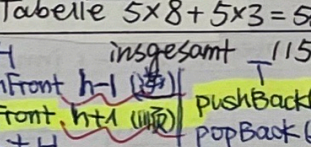
Durchschnitt-Code-Länge

$= (\sum$ jeweils Count · jeweil Code-Bits$) /$ Gesamt-Count

$= (3 \times 3 + 5 \times 2 + 6 \times 2 + 4 \times 2 + 2 \times 3) / 20 = 45/20 = 2{,}25$ Bits

Shannon-Fano: Baum wird von Wurzel abwärts erstellt (与Hoffman相反)

Adjazenz-Array



L'Hôpital rule:

if $\lim \frac{f(x)}{g(x)} = 0$ or $\lim \frac{f(x)}{g(x)} = \frac{\infty}{\infty}$, then $\lim \left|\frac{f'(x)}{g'(x)}\right|$

$\sup < \infty \Rightarrow f(n) \in O(g(n))$ $\sup = 0 \Rightarrow f(n) \in o(g(n))$

Lim sup = Obergrenze von Limes in Unendlichen

Lim inf = Untergrenze von Limes in Unendlichen

$\inf > 0 \Rightarrow f(n) \in \Omega(g(n))$ $\inf = \infty \Rightarrow f(n) \in \omega(g(n))$

$f'(\ln n) = \frac{1}{n}$

Quicksort

median of three: 取 first, middle, last 中的中位数作为 pivot 只要不是 last 作 pivot 开始前都要与 last 交换

i 找到 a[i] > pivot 就停止, 否则 i++

j 找到 a[j] < pivot 就停止, 否则 j--

⇒ 当 j 指针停止, 就 swap a[i] 和 a[j]

如果 i ≥ j 就停止, swap a[j] 和 pivot

⇒ $f(n) \in o(g_n)$

```
pushFront(x){ H=(H-1)%(n+1); array[H]=x; }
pushBack(x){ array[t]=x; t=(t+1)%(n+1); }
```

pushFront h-1 (减)

popFront h+1 (减)

pushBack(x) t+1 (加)

popBack t-1 (减)

% (n+1)