Effiziente Algorithmen

Zusammenfassung der Vorlesung im Frage-Antwort-Stil
Krefeld, Dezember 2021
Antworten sind nicht als Musterlösung zu verstehen! Keine Hilfsmittel in Klausur erlaubt!

1 Berechenbarkeit und Komplexitätstheorie

Frage: Welche Eigenschaften interessieren uns bei Algorithmen?

Antwort: Korrektheit, Laufzeit, Speicherplatz, Kommunikationszeit, Güte u.a.

Frage: Warum ist es keine gute Idee, zum Bewerten von Algorithmen deren Laufzeiten zu messen?

Antwort: Laufzeiten messen und vergleichen bringt nichts aufgrund unterschiedlich schneller Hardware, unterschiedlich guter Compiler, unterschiedlicher Betriebssysteme und Laufzeitumgebungen, unterschiedlichen Eingabedarstellungen und Datenstrukturen usw.

Frage: Wie bewerten bzw. vergleichen wir Algorithmen?

Antwort:

- Wir verwenden idealisierte Rechenmodelle wie die Registermaschine (Random Access Machine, RAM): festgelegter Befehlssatz (Assembler-ähnlich), abzählbar unendlich viele Speicherzellen.
  Die Laufzeit ist dann die Anzahl ausgeführter RAM-Befehle, der Speicherbedarf ist die Anzahl der benötigten Speicherzellen.
- Bei den einzelnen Problemstellungen ermitteln wir charakteristische Parameter. Beim Sortieren sind dies die Anzahl der Schlüsselvergleiche bzw. Vertauschungen, bei arithmetischen Problemen bspw. die Anzahl der Additionen oder Multiplikationen.

Laufzeit und Speicherbedarf sind in der Regel abhängig von der Größe der Eingabe.

Frage: Warum ist die Komplexität der Algorithmen interessant? Sind heutige Rechner nicht für alle Probleme ausreichend schnell?

Antwort: Weil bei großen Laufzeiten weder schnellere Rechner noch verteilte bzw. parallele Rechner eine signifikante Laufzeitverbesserung bringen. Bei exponentieller Laufzeit wie 2^n kann ein doppelt so schneller Rechner nur eine um eins größere Eingabe in der gleichen Zeit bearbeiten, da $2 \cdot 2^n = 2^{n+1}$ ist. Analog verhalten sich parallele Algorithmen, wo im Mittel nie mehr als ein linearer Speedup erreicht werden kann.

Frage: Wir haben zur asymptotischen Aufwandsabschätzung die Klasse Groß-O definiert. Geben Sie diese Definition an.

Antwort: $\mathcal{O}(g) = \{f \mid \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$ Dabei betrachten wir nur Funktionen mit natürlichen Funktionswerten, also $f, g : \mathbb{N} \to \mathbb{N}$, weil wir die Anzahl von Schritten oder benutzten Speicherzellen angeben.

Frage: Wird die Groß-O-Notation nur für die Angabe von worst-case-Abschätzungen genutzt? Mit Erläuterung!

Antwort: Nein. Es können auch obere Schranken für average- oder best-case Abschätzungen angegeben werden. Groß-O heißt nicht automatisch worst-case.

Frage: Wie ist die Worst-Case-Laufzeitkomplexität definiert?

Antwort: Die Menge der zulässigen Eingaben der Länge n bezeichnen wir mit W_n, die Anzahl der Schritte von Algorithmus A für Eingabe w mit $A(w)$.

Dann ist die Worst-Case-Komplexität definiert als $T_A(n) = \sup\{A(w) \mid w \in W_n\}$ und ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten.

Frage: Was bezeichnen wir als Sprache?

Antwort: Ein *Alphabet* ist eine endliche, nichtleere Menge von Zeichen. Zeichen werden auch Symbole oder Buchstaben genannt.

Endliche Folgen $(x_1, \ldots, x_k)$ mit $x_i \in A$ heißen *Wörter* der Länge k über dem Alphabet A. Die Menge aller Wörter über einem Alphabet A wird mit $A^\star$ bezeichnet. Das leere Wort wird durch das Symbol ϵ dargestellt.

Eine Teilmenge von $A^\star$ wird als *Sprache* bezeichnet.

Frage: Wir haben als Rechenmodell auch die Turingmaschine kennengelernt. Beschreiben Sie dieses Rechenmodell.

Eine dTM besteht aus einem unendlich langen Band, einem Schreib-/Lesekopf und einer Zustandssteuerung:

- Das potentiell unendliche Band ist in Felder unterteilt, wobei jedes Feld ein einzelnes Zeichen des Arbeitsalphabets der Maschine enthalten kann.
- Auf dem Band kann sich der Schreib-Lesekopf bewegen, wobei nur das Zeichen, auf dem sich dieser Kopf gerade befindet, im momentanen Rechenschritt verändert werden kann.
- Der Kopf kann in einem Rechenschritt nur um maximal eine Position nach links oder rechts bewegt werden.
- Noch nicht besuchte Felder enthalten das Blank-Symbol B.

Frage: Ist die Klasse der entscheidbaren (rekursiven) Sprachen abgeschlossen gegen Komplementbildung? Mit Erklärung!

Antwort: Ja. Aus akzeptierenden Endzuständen von M werden nicht-akzeptierende Endzustände von M' und man führt einen neuen akzeptierenden Endzustand ein, der immer dann erreicht wird, wenn die ursprüngliche Maschine M in einem nicht-akzeptierenden Endzustand hält.

Frage: Kann eine 2-Band TM mehr berechnen als eine 1-Band TM? Mit Begründung!

Antwort: Nein. Jede $t(n)$-zeit- und $s(n)$-platzbeschränkte k-Band TM kann durch eine $\mathcal{O}(t(n) \cdot s(n))$-zeit- und $\mathcal{O}(s(n))$-platzbeschränkte TM simuliert werden.

Beweisidee: Verwende 2k Spuren. Auf einem Spurpaar steht der Inhalt eines Bandes und ein Marker für die Position des Schreib-/Lesekopfes. Um einen Schritt der k-Band TM zu simulieren: Laufe einmal über das gesamte Band, um die Position der Schreib-/Leseköpfe zu finden. Merke im Zustand die Inhalte der Bänder und führe den Zustandsübergang durch. Ändere beim Zurücklaufen über das Band die Marker für die Kopfpositionen.

Frage: Wann bezeichnen wir eine Funktion als berechenbar?

Antwort: Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist berechenbar, falls es ein RAM-Programm P gibt (Achtung: die Existenz reicht aus!), sodass gilt: P berechnet m zur Eingabe $n_1, \ldots, n_k$ genau dann wenn $f(n_1, \ldots, n_k) = m$ gilt.

Frage: Gibt es Funktionen, die nicht berechenbar sind? Beschreiben Sie mindestens zwei dieser nicht berechenbaren Sprachen bzw. Funktionen.

Antwort: Diagonalsprache:

$$DIAG = \{\langle M \rangle \mid M \text{ ist dTM, die } \langle M \rangle \text{ nicht akzeptiert}\}$$

Halteproblem: Stoppt ein Programm zu einer bestimmten Eingabe?

$$H = \{\langle M \rangle x \mid M \text{ ist dTM, die gestartet mit Eingabe } x \text{ hält}\}$$

Halteproblem bei leerem Band:

$$H_0 = \{\langle M \rangle \mid M \text{ ist dTM, die gestartet mit Input } \epsilon \text{ hält}\}$$

Totalitätsproblem:

$$\{\langle M \rangle \mid M \text{ hält für jeden Input}\}$$

Endlichkeitsproblem:

$$\{\langle M \rangle \mid M \text{ hält für endlich viele Inputs}\}$$

Äquivalenzproblem:

$$\{\langle M \rangle, \langle M' \rangle \mid M \text{ und } M' \text{ akzeptieren die gleiche Sprache}\}$$

Postsches Korrespondenzproblem: Gegeben ist eine endliche Menge von Wortpaaren (x_1, y_1), $(x_2, y_2), \ldots, (x_k, y_k)$, wobei $x_i, y_i \in \Sigma^+$ gilt. Gibt es eine endliche Folge von Indizes $i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, k\}$, $n \geq 1$, mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$?

Frage: Wodurch unterscheiden sich dTMs von nTMs?

Antwort: nTMs besitzen anstelle der Übergangsfunktion eine Übergangsrelation, d.h. jede Konfiguration hat mehrere mögliche Nachfolgekonfigurationen. Der Akzeptanzbegriff ist anders definiert: Eine nTM akzeptiert eine Eingabe gdw. eine Berechnung von der Startkonfiguration in eine akzeptierende Endkonfiguration existiert.

Frage: Wird der Berechenbarkeitsbegriff durch den Nichtdeterminismus erweitert? Mit Erklärung.

Antwort: Nein, eine polynomiell zeitbeschränkte nTM kann durch eine exponentiell zeitbeschränkte dTM simuliert werden.

Simuliere Maschine, die jeweils maximal b mögliche Zustandsübergänge ermöglicht und betrachte die Abarbeitung einer Eingabe als Baum. Die Idee der Simulation: Teste nacheinander

alle möglichen Berechnungswege. Dazu werden auf einem Band alle möglichen Verzweigungen aufgezählt, ein anderes Band enthält den Bandinhalt der zu simulierenden Maschine.

Frage: Wie haben wir eine erste, nicht berechenbare Funktion gefunden? Mit Erklärung.

Antwort: Mittels Diagonalisierung. Sei $M_1, M_2, \ldots$ die Folge aller in der Reihenfolge ihrer Gödelnummern aufgezählten dTMs.

Betrachte die unendliche Matrix, deren Zeilen mit $\langle M_1 \rangle, \langle M_2 \rangle, \ldots$, und deren Spalten mit $M_1, M_2, \ldots$ nummeriert sind.

An Position M_i, $\langle M_j \rangle$ wird eingetragen, ob M_i die Eingabe $\langle M_j \rangle$ akzeptiert (Eintrag „a“) oder nicht akzeptiert (Eintrag „na“).

	M_1	M_2	M_3	$\cdots$
$\langle M_1 \rangle$	a	na	a	$\cdots$
$\langle M_2 \rangle$	na	na	a	$\cdots$
$\langle M_3 \rangle$	na	na	a	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Spalte unter M_i heißt *Akzeptanz-Folge* von M_i. Wir nehmen die Existenz einer dTM $\bar{M}$ an, die $DIAG$ akzeptiert.

$$DIAG = \{\langle M \rangle \mid M \text{ ist dTM, die } \langle M \rangle \text{ nicht akzeptiert}\}$$

Wie sieht die Akzeptanz-Folge von $\bar{M}$ aus?

- Definition von $DIAG$: An Stelle i hat sie den Eintrag a falls M_i die Eingabe $\langle M_i \rangle$ nicht akzeptiert, sonst den Eintrag na.
- Somit ist $\bar{M}$ keine der dTMs $M_1, M_2, \ldots$, da sich die Akzeptanz-Folge von M_i an Stelle i von der von $\bar{M}$ unterscheidet.
- Somit kann es $\bar{M}$ nicht geben, da wir ja in $M_1, M_2, \ldots$ alle dTMs aufgezählt haben. Widerspruch!

Frage: Welche Beweistechnik kennen Sie noch, um zu zeigen, dass eine Funktion nicht berechenbar ist? Mit Erklärung.

Antwort: Reduktion. Eine Sprache L heißt reduzierbar auf L', falls es eine totale berechenbare Funktion $f : \Sigma^* \longrightarrow \Sigma^*$ gibt mit $x \in L \iff f(x) \in L'$.

- wir schreiben dann: $L \leq L'$ (mittels f)
- Idee: L ist nicht schwerer zu lösen als L'

Falls L nicht entscheidbar ist und $L \leq L'$ (mittels f) gilt, dann ist auch L' nicht entscheidbar. Denn wäre L' entscheidbar, dann wäre auch L entscheidbar:

- Berechne für eine beliebige Eingabe x für L den Wert $f(x)$
- und entscheide, ob $f(x) \in L'$ gilt.
- Damit wäre auch entschieden, ob $x \in L$ ist.

Frage: Warum betrachten wir immer die Entscheidungsvariante und nicht die Optimierungsversion der Probleme, wenn es um NP-Vollständigkeit geht?

Antwort: Bei der Optimierungsvariante reicht es nicht, die „geratene“ Lösung zu verifizieren. Es muss auch sichergestellt werden, dass es keine bessere Lösung gibt.

Frage: Wie ist NP-vollständig definiert?

Antwort: Ein Problem L ist NP-vollständig, wenn gilt:

- Es existiert eine nTM, die das Problem löst, also $L \in \text{NP}$
- und alle L' aus NP sind polynomiell reduzierbar auf L, also $L' \leq_p L$.

Das besondere dabei: Ist ein NP-vollständiges Problem effizient (d.h. in polynomieller Zeit) lösbar, dann sind automatisch alle NP-vollständigen Probleme effizient lösbar! (Mittels der bei den Reduktionen angegebenen Reduktionsfunktionen.) Aber obwohl gerade diese Probleme in der Industrie oft zu lösen sind und daher ein riesiges Interesse an guten Algorithmen existiert, sind bisher keine effizienten Algorithmen gefunden worden. Man vermutet daher, dass es solche Algorithmen nicht gibt.

Frage: Was heißt polynomiell reduzierbar?

Antwort: Eine Sprache L ist *polynomiell reduzierbar auf* eine Sprache L', wenn eine Funktion f existiert, mit

- $x \in L \iff f(x) \in L'$
- und f ist in polynomieller Zeit berechenbar.

Frage: Welches war das erste Problem, von dem man zeigen konnte, dass es NP-vollständig ist? Skizzieren Sie den Beweis.

Antwort: Das Erfüllbarkeitsproblem (SATisfiability):

- gegeben: eine Formel F der Aussagenlogik
- gefragt: Ist die Formel F erfüllbar?

Wir müssen eine Boolesche Formel F angeben, so dass gilt:

$$x \in L \iff F \text{ ist erfüllbar}$$

Die gesuchte Formel F enthält folgende boolesche Variablen:

Variable	Indizes	Bedeutung
$zust_{t,z}$	$t = 0, \ldots, p(n)$ und $z \in Z$	$zust_{t,z} = 1 \iff$ nach t Schritten befindet sich M im Zustand z
$pos_{t,i}$	$t = 0, \ldots, p(n)$ und $i = -p(n), \ldots, p(n)$	$pos_{t,i} = 1 \iff$ der Schreib-Lesekopf von M befindet sich nach t Schritten auf Position i
$band_{t,i,a}$	$t = 0, \ldots, p(n)$ und $i = -p(n), \ldots, p(n)$ und $a \in \Gamma$	$band_{t,i,a} = 1 \iff$ nach t Schritten befindet sich auf Bandposition i das Zeichen a

Die Formel F ist aus mehreren Teilformeln aufgebaut, die unterschiedliche Dinge repräsentieren:

- Randbedingungen: Die TM M ist zu jedem Zeitpunkt t in genau einem Zustand; der Kopf von M befindet sich zu jedem Zeitpunkt t an genau einer Bandposition; zu jedem Zeitpunkt t und an jeder Bandposition i steht genau ein Zeichen.

- Anfangsbedingung: Die TM M ist im Anfangszustand z_0; der Schreib-/Lesekopf befindet sich auf dem ersten Zeichen; die ersten n Bandpositionen enthalten die Eingabe $x_1, \ldots, x_n$; an allen anderen Bandpositionen steht ein Blank.
- Beim Übergang vom Zeitpunkt t nach $t+1$ darf sich an der Stelle, wo sich der Schreib-Lesekopf befindet, der Bandinhalt ändern; an allen anderen Stellen darf sich der Bandinhalt nicht ändern.
- Endbedingungen: Ist die Maschine M in einem akzeptierenden Zustand?

Frage: Definieren Sie das Vertex-Cover-(Entscheidungs-)Problem. Geben Sie insbesondere an, was gegeben ist und was gesucht ist.

Antwort: Finde zu einem Graphen $G = (V, E)$ und einer Zahl $k \in \mathbb{N}$ eine Knotenmenge $V' \subseteq V$ mit $|V'| \leq k$, sodass jede Kante $e \in E$ inzident zu einem Knoten in V' ist.

Frage: Beschreiben Sie zur Lösung des Vertex-Cover-(Entscheidungs-)Problems den Brute-Force-Ansatz und schätzen Sie die Laufzeit dieses naiven Verfahrens ab.

Antwort: Suche ein Vertex Cover, indem alle $\binom{n}{k}$ vielen Teilmengen der Größe k betrachtet werden. Laufzeit: $\mathcal{O}(\binom{n}{k} \cdot |G|) \in \mathcal{O}(n^k \cdot |G|)$

Frage: Beschreiben Sie eine Faktor-2-Approximation für das Problem Vertex-Cover (in der Optimierungsvariante) und begründen Sie die Fehlerschranke.

Antwort:

$C := \emptyset$
while es gibt noch Kanten in G **do**
    nimm irgendeine Kante $\{u, v\}$ von G
    nimm u und v beide in C auf
    entferne u und v und alle dazu inzidenten Kanten aus G

Sei F die Menge der ausgewählten Kanten, sei C das berechnete Vertex-Cover und sei $\{u, v\} \in F$. Jedes Vertex-Cover C' muss u oder v enthalten, da sonst die Kante $\{u, v\}$ nicht abgedeckt würde. Also gilt:

$$|C'| \geq \frac{1}{2}|C| \iff |C| \leq 2|C'|$$

Frage: Definieren Sie, wann ein parametrisiertes Problem (Π, k) fixed-parameter-tractable ist. Dabei ist Π ein Entscheidungsproblem, $\mathcal{I}$ die Menge aller möglichen Eingaben und $k : \mathcal{I} \to \mathbb{N}$ der Parameter.

Antwort: Ein Algorithmus A ist ein *fpt-Algorithmus*, wenn die Laufzeit für jede Eingabe $I \in \mathcal{I}$ in $\mathcal{O}(f(k(I)) \cdot |I|^c)$ liegt, wobei f eine Funktion und $c \in \mathbb{N}$ konstant ist. Ein parametrisiertes Problem (Π, k) gehört zur Klasse *FPT* und wird *fixed parameter tractable* genannt, wenn es einen fpt-Algorithmus bezüglich k gibt, der Π entscheidet.

Frage: Geben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(2^k \cdot (n+m))$ für das Vertex-Cover-Problem an, wobei n die Anzahl der Knoten und m die Anzahl der Kanten des Graphen ist.

Antwort: binärer Entscheidungsbaum mit beschränkter Tiefe k:

function VC(G, k)
    if $k = 0$ und es gibt noch Kanten in G **then return** false

if es gibt keine Kanten in G **then return** true
nimm irgendeine Kante $\{u, v\}$ von G
$x := \mathrm{VC}(G - \{u\}, k-1)$
$y := \mathrm{VC}(G - \{v\}, k-1)$
return $x \vee y$

Laufzeit: $\mathcal{O}(2^k \cdot |G|)$

Frage: Betrachten Sie den folgenden Algorithmus. Sei Φ eine Formel in konjunktiver Normalform, bei der jede Klausel höchstens 3 Literale enthält.

```
3SAT(Φ):
    if 3SAT(Φ|x)
        return true
    return 3SAT(Φ|¬x)
```

Dabei bedeutet $\Phi|x$, dass die Variable x der Formel Φ mit 1 belegt ist. Schätzen Sie die Laufzeit ab (mit Herleitung) und begründen Sie, warum der Algorithmus korrekt ist.

Antwort: Variable x muss entweder mit 0 oder 1 belegt werden. Da alle 2^n mögliche Belegungen getestet werden, ist der Algorithmus korrekt.

$T(n) = 2 \cdot T(n-1) + poly(n) \in \mathcal{O}^*(2^n)$

$$\begin{array}{rcll} b^n & \approx & 2 \cdot b^{n-1} + n^k & | \ : b^{n-1} \\ \iff b & \approx & 2 + \frac{n^k}{\cancel{b^{n-1}}} & | \ \lim\limits_{n\to\infty} \frac{n^k}{b^{n-1}} = 0 \\ \iff b & \approx & 2 & \end{array}$$

2 Entwurfsmethoden

Frage: Welche grundlegenden Entwurfsmethoden für Algorithmen kennen Sie?

Antwort: divide and conquer, dynamic programming, greedy und local search.

Frage: Wie funktioniert „divide and conquer“?

Antwort:

- *Divide* the problem into subproblems.
- *Conquer* the subproblems by solving them recursively.
- *Combine* subproblem solutions.

Frage: Als Beispiel für einen Divide-And-Conquer-Algorithmus hatten wir Strassens Matrixmultiplikation kennengelernt. Wodurch unterscheidet sich der vom naiven Algorithmus?

Antwort: anstatt die jeweils 4 Teilmatrizen nach dem Schema

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \qquad \begin{array}{rcl} r & = & ae + bg \\ s & = & af + bh \end{array} \qquad \begin{array}{rcl} t & = & ce + dg \\ u & = & cf + dh \end{array}$$

mit 8 Unter-Matrixmultiplikationen zu berechnen, hat Strassen eine Berechnung mit nur 7 Unter-Matrixmultiplikationen gefunden. Daher ergibt sich als Laufzeit anstatt von

$$T(n) = 8 \cdot T(n/2) + \mathcal{O}(n^2) \in \Theta(n^3)$$

bei der naiven Methode bei Strassens Methode nur eine Laufzeit von

$$T(n) = 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \in \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.807}).$$

Frage: Welche Idee liegt der „dynamischen Programmierung“ zugrunde?

Antwort: Bei Divide-And-Conquer entstehen in den rekursiven Aufrufen Teillösungen, die wir im Combine-Schritt zusammen fügen müssen. Damit wir nicht dieselben Teilprobleme immer wieder rekursiv lösen, werden bereits berechnete Teillösungen in Tabellen gespeichert. Diesen Ansatz nennt man memorieren.

Anstatt die Teillösungen rekursiv zu berechnen, kann man aber auch einen Bottom-Up-Ansatz wählen: Aus Rekursion wird Iteration, indem wir aus kleinen Teillösungen größere Lösungen zusammensetzen. Da wir nicht von vorneherein wissen, welche Teillösungen wir benötigen, berechnen wir einfach alle. Dieses Verfahren wird oft bei Optimierungsproblemen eingesetzt.

Frage: Bei welchen Problemen haben wir dynamische Programmierung eingesetzt?

Antwort: Bei der Matrix-Kettenmultiplikation, beim 0/1-Rucksack-Problem, bei der Polygon Triangulierung, der Levenshtein-Distanz und bei einigen Graphalgorithmen wie z.B. die transitive Hülle oder all-pairs-shortest-path.

Frage: Welche allgemeine Bedingung muss erfüllt sein, damit dynamische Programmierung eingesetzt werden kann?

Antwort: Das Optimalitätsprinzip von Bellman: Eine optimale Lösung kann aus optimalen Teillösungen zusammengesetzt werden. Gilt z.B. bei kürzesten Wegen.

Frage: Geben Sie die zu optimierende Funktion bei der Matrix-Kettenmultiplikation an.

Antwort: Bezeichne $m(i, j)$ die minimale Anzahl skalarer Multiplikationen bei optimaler Klammerung des Abschnitts $M_i \cdots M_j$.

Sei $(M_i \cdots M_k) \cdot (M_{k+1} \cdots M_j)$ die optimale Klammerung des Abschnitts $M_i \cdots M_j$. Dann gilt:

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$$

Damit ergibt sich die Rekursionsformel:

$$m(i, j) = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} (m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j), & i < j \end{cases}$$

Frage: Geben Sie die zu optimierende Funktion beim 0/1-Rucksack-Problem an.

Antwort: Sei $knap(h, i)$ der maximale Wert, der mit den Objekten $i, \ldots, n$ und Rucksackgröße h erreicht werden kann. Dann gilt:

$$knap(h, i) = \max\{knap(h, i + 1),\ w_i + knap(h - g_i, i + 1)\}$$

Als Rekursionsende ergibt sich:

$$knap(h,i) = \begin{cases} knap(h,i+1), & i < n, h < g_i \\ 0, & i = n, h < g_n \\ w_n, & i = n, h \geq g_n \end{cases}$$

Frage: Welche Idee liegt der Greedy-Methode zugrunde und welche Probleme haben wir mit Greedy-Algorithmen gelöst?

Antwort: Es werden rein lokale Entscheidungen getroffen, bei denen nach einem gegebenen Kostenmaß das jeweils nächste Element gewählt wird. Die Korrektheit der Lösungen muss gezeigt werden.

Probleme: Optimal Merge Patterns, Rucksack-Problem, Wechselgeld-Problem, sowie Präfix-Code nach Huffman und bei den Graphalgorithmen z.B. Minimaler Spannbaum nach Kruskal.

Frage: Idee bei Optimal Merge Patterns?

Antwort: Mische immer die beiden Dateien, die die geringste Länge haben.

Frage: Idee beim Rucksack-Problem?

Antwort: Nimm als nächstes das Objekt in den Rucksack auf, das den größten Nutzen pro Gewichtseinheit besitzt. Problem hierbei: Das letzte Objekt kann evtl. nur teilweise in den Rucksack aufgenommen werden. Dies ist also keine Lösung für das 0/1-Rucksack-Problem.

Frage: Idee beim Wechselgeld-Problem?

Antwort: Nimm möglichst wenige Münzen, indem der Betrag zunächst um ein Vielfaches der größten Münze reduziert wird. Problem hierbei: Funktioniert nicht bei allen Münzfolgen: Bei den Münzen 11,7,1 wird der Betrag 14 als 11,1,1,1 ausgezahlt, aber 7,7 wäre besser.

Frage: Wie kann mittels dynamischer Programmierung das Wechselgeld-Problem gelöst werden?

Antwort: Bezeichne $d_1, \ldots, d_k$ die Werte der Münzen und $coins(p)$ die minimale Anzahl Münzen, um den Betrag p auszuzahlen. Dann ergibt sich folgende Optimierungsfunktion:

$$coins(p) = \begin{cases} 0, & \text{falls } p = 0 \\ \min\limits_{i:d_i \leq p} \{1 + coins(p - d_i)\}, & \text{sonst} \end{cases}$$

Frage: Welche Idee liegt der lokalen Suche zugrunde?

Antwort: Mutiere eine beliebige Startlösung durch geringfügige, lokale Veränderungen. Übernehme eine neue Lösung, falls diese besser ist als die alte Lösung. Um lokale Optima zu vermeiden, starte mit verschiedenen initialen Lösungen.

3 Sortieren

Frage: Wie funktioniert Quicksort?

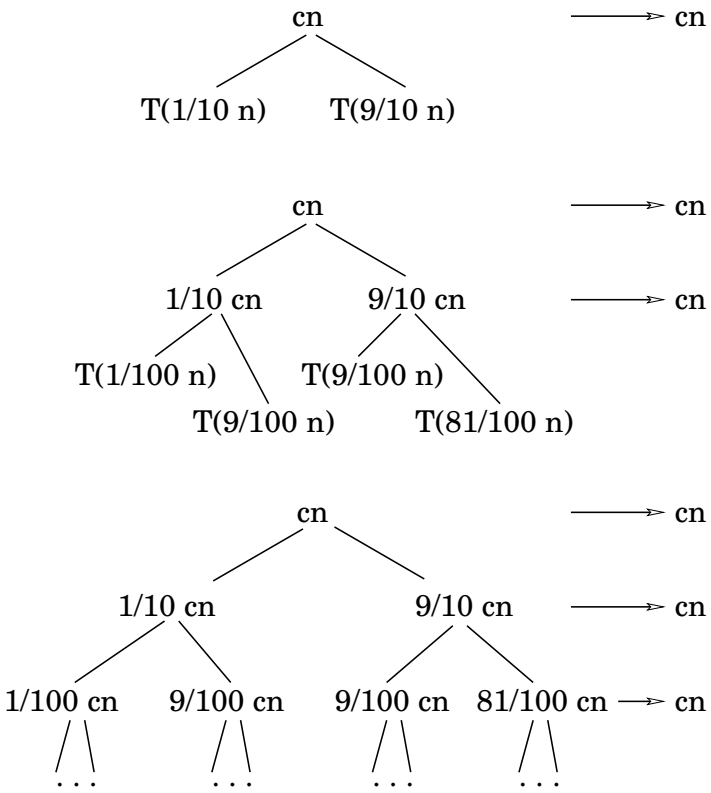
Antwort: Es ist ein Divide-And-Conquer Algorithmus: Teile die gegebene Folge in zwei Teilfolgen L und R mittels eines Pivot-Elements p auf, und sortiere die Teilfolgen rekursiv: sorted(L), p, sorted(R)

Frage: Wie funktioniert die Partitionierung von Quicksort?

Antwort: Wähle ein Pivot-Element p. Suche das erste Element (von links), das größer als p ist, und suche das erste Element (von rechts), das kleiner als p ist, und vertausche die beiden Elemente. Wiederhole das Suchen und Vertauschen. Tausche zum Schluss das Pivot-Element an die richtige Stelle.

Frage: Welche Laufzeit hat Quicksort, wenn die Aufteilung immer in einem festen Verhältnis erfolgt, z.B. immer im Verhältnis 9:1?

Antwort: $O(n \cdot \log(n))$. Eine der resultierenden Folgen hat die Länge $\frac{1}{10}n$, die andere eine Länge von $\frac{9}{10}n$. Damit ergibt sich als Laufzeit:



Binomische Formel:

$$(a+b)^k \;=\; \sum_{i=0}^{k} \binom{k}{i} a^{k-i} \cdot b^i$$

Daher gilt:

$$\begin{aligned} T(n) \;&=\; \sum_{i=0}^{k} \binom{k}{i} \frac{9^i}{10^k} \cdot cn \;=\; \frac{cn}{10^k} \sum_{i=0}^{k} \binom{k}{i} 9^i \\ &=\; \frac{cn}{10^k} \sum_{i=0}^{k} \binom{k}{i} 1^{k-i} \cdot 9^i \;=\; \frac{cn}{10^k} \cdot (1+9)^k \;=\; cn \end{aligned}$$

Da der obige Baum logarithmische Tiefe hat, ergibt sich als Laufzeit $O(n \cdot \log(n))$.

Frage: Welche Laufzeit hat Quicksort im mittleren Fall? Können Sie skizzieren, wie man zu diesem Ergebnis kommt?

Antwort: Die Wahrscheinlichkeit, dass die Folge an Position p aufgeteilt wird, ist $1/n$. Das Aufteilen in zwei Teilfolgen erfolgt in Zeit $O(n)$.

$$T(n) \;=\; c \cdot n \;+\; \frac{1}{n} \cdot \sum_{p=1}^{n} (T(p-1) + T(n-p))$$

Durch umformen ergibt sich schließlich

$$T(n) \;\in\; \Theta(n \cdot \log(n))$$

Frage: Welche worst-case Laufzeit hat Quicksort und für welche Folgen wird diese tatsächlich erreicht?

Antwort:

- worst-case Laufzeit ist $O(n^2)$
- stark vorsortierte Folgen oder solche Folgen, die viele gleiche Elemente enthalten

---

Frage: Wie können wir Quicksort verbessern für den Fall, dass die Folgen stark vorsortiert sind?

Antwort: Zufallsstrategie: wähle als Pivot-Element ein zufälliges Element aus $A[l...r]$ und vertausche es mit $A[l]$.

$\Rightarrow$ Laufzeit ist unabhängig von der zu sortierenden Folge
$\Rightarrow$ mittlere/erwartete Laufzeit: $\Theta(n \cdot \log(n))$

Frage: Wie können wir Quicksort verbessern für den Fall, dass viele gleiche Werte zu sortieren sind?

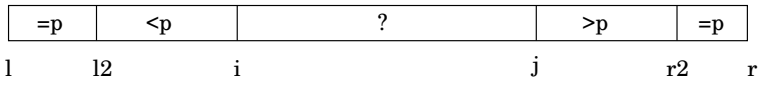
Antwort: 3-Wege-Split Quicksort:

Teile die Folge $a[l], \ldots, a[r]$ in drei Folgen F_l, F_m, F_r auf.

- F_l enthält die Elemente mit Schlüssel $< k$.
- F_m enthält die Elemente mit Schlüssel $= k$.
- F_r enthält die Elemente mit Schlüssel $> k$.

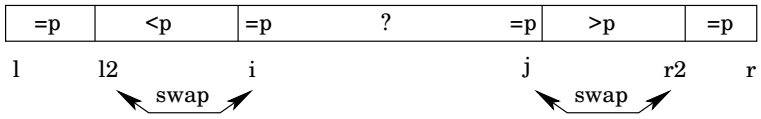
Sortiere F_l und F_r auf dieselbe Weise.

Idee: Die Pivotelemente werden zuerst am Rand gesammelt (zwischen l und l2 sowie zwischen r2 und r) und vor dem rekursiven Aufruf in die Mitte transportiert.

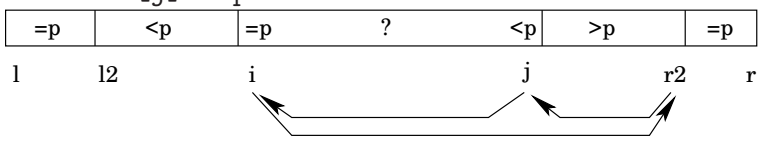


während der Aufteilung (Partitionierung) sind drei Fälle zu unterscheiden, der vierte Fall wird wie im ursprünglichen Quicksort behandelt:

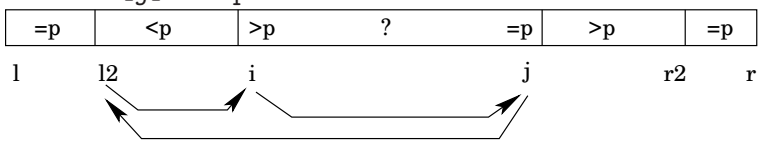
Fall 1: `a[i] == p` und `a[j] == p`



Fall 2: `a[i] == p` und `a[j] < p`



Fall 3: `a[i] > p` und `a[j] == p`



Frage: Wie groß ist die Rekursionstiefe bei Quicksort im worst-case und wie können wir die Rekursionstiefe beschränken?

Antwort: Im ungünstigen Fall ist die Rekursionstiefe $O(n)$.

Wir können die Rekursionstiefe aber auf $O(\log(n))$ beschränken:

- löse das kleinere Teilproblem rekursiv und
- löse das größere Teilproblem iterativ direkt.

```
quicksort(int l, int r)
    while l < r
        m := partition(l, r)
        if (m-l) < (r-m) then
            quicksort(l, m-1)
            l := m + 1
        else
            quicksort(m+1, r)
            r := m - 1
```

Frage: Welche Sortierverfahren sortieren im worst-case in Zeit $O(n \cdot \log n)$?

Antwort: Mergesort und Heapsort.

Frage: Können Sie Mergesort skizzieren und die Rekursionsgleichung angeben?

Antwort:

- Teile die Folge in zwei etwa gleich große Teilfolgen.
  Sortiere die Teilfolgen rekursiv.
  Mische die sortierten Teilfolgen zu einer Folge zusammen.
- $T(n) = 2 \cdot T(n/2) + c \cdot n \in O(n \cdot \log n)$

Frage: Können Sie Heapsort skizzieren und die Laufzeit herleiten? Wie lange dauert das Aufbauen eines initialen Heaps und wie geht das?

Antwort:

- Erstelle einen initialen Max-Heap. Anschließend wird in Runde k das oberste Element mit dem Element an Position $n - k$ getauscht und das nun oberste Element versickert.
- Es werden n Runden benötigt, jeweils mit der Laufzeit $O(\log n)$, da der Heap nur logarithmische Tiefe hat. Aufbauen des initialen Heaps erfolgt in Zeit $O(n)$.
- Das Erstellen des initialen Heaps erfolgt, indem wir die Elemente $k_{n/2-1}, \ldots, k_0$ nacheinander versickern lassen. Dabei müssen $n/4$ Elemente nur um eine Ebene im Baum verschoben werden, $n/8$ viele Elemente nur um 2 Ebenen usw.

Frage: Was ist ein allgemeines Sortierverfahren und welche untere Schranke gilt für solche allgemeinen Sortierverfahren?

Antwort: Allgemeine Sortierverfahren verwenden nur Vergleichsoperationen zwischen Schlüsseln, keine arithmetischen Operationen oder ähnliches.

Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln im schlechtesten Fall und im Mittel wenigstens $\Omega(n \cdot \log(n))$ Schlüsselvergleiche.

Frage: Können Sie das skizzieren?

Antwort: Man zeigt das über die Höhe von Entscheidungsbäumen.

Worst-Case-Analyse:

- es gibt $n!$ verschiedene Permutationen über n Zahlen
- ein Entscheidungsbaum hat mindestens $n!$ Blätter
- ein Binärbaum der Höhe h hat maximal $2^h - 1$ Blätter
- es muss also gelten: $2^h \geq n!$

$$
\begin{array}{rcll}
h & \geq & \log(n!) & \text{log ist monoton steigend} \\
 & \geq & \log((n/e)^n) & \text{Stirling-Formel} \\
 & = & \log(n^n) - \log(e^n) & \\
 & = & n \cdot \log(n) - n \cdot \log(e) & \\
 & \in & \Omega(n \cdot \log(n)) &
\end{array}
$$

Frage: Können wir schneller als in Zeit $O(n \cdot \log(n))$ sortieren?

Antwort: Ja, mittels Counting-Sort bzw. Radix-Sort. Die funktionieren aber nur gut für Integer-Werte. Zusammengesetzte Datentypen wie `double` (setzt sich aus Mantisse und Exponent zusammen) bereiten bei Radix-Sort Probleme, Counting-Sort ist nicht brauchbar bei großen Wertebereichen.

4 Auswahlproblem

Frage: Was ist das Auswahlproblem und welche zwei Lösungsansätze haben wir betrachtet?

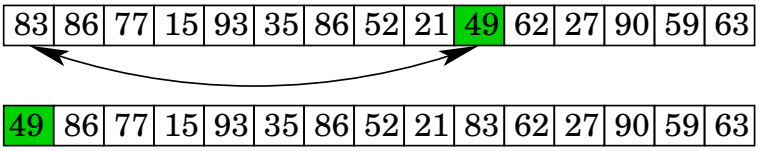
Antwort: Beim Auswahlproblem soll das i-kleinste Element in einer unsortierten Folge gefunden werden. Ein Spezialfall ist die Bestimmung des Medians.

Betrachtet haben wir die randomisierte Selektion und die Median-der-Mediane-Strategie.

Frage: Wie funktioniert die randomisierte Selektion und welche Laufzeit hat sie?

Antwort: `random-select(l,r,i)` basiert auf der Idee von Quicksort:

- Wähle aus allen Werten ein Pivot-Element zufällig aus und tausche es mit dem Element am linken Rand



- `k := partition(l,r)`: teile die Folge in zwei Teilfolgen L und R auf

- $i < k \Rightarrow$ gib `random-select(l,k-1,i)` zurück
- $i = k \Rightarrow$ gib das Pivot-Element zurück
- $i > k \Rightarrow$ gib `random-select(k+1,r,i)` zurück

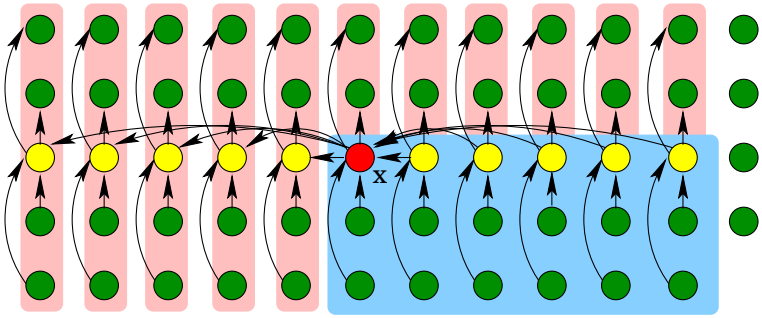
Die Partitionierung erfolgt in Zeit $\Theta(n)$, daher ergibt sich als Laufzeit:

- *lucky:* $T(n) = T(9n/10) + \Theta(n) = \Theta(n)$
- *unlucky:* $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Frage: Wie funktioniert die Median-der-Mediane-Strategie und welche Laufzeit hat sie?

Antwort: `SELECT(l,r,i)`

1. divide the $r - l + 1$ elements into groups of 5
2. find the median of each 5-element group by rote
3. recursively select the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot
4. partition around the pivot x and let `k := partition(l,r)`
5. if $i < k$ return `SELECT(l,k-1,i)`
6. if $i = k$ return pivot
7. if $i > k$ return `SELECT(k+1,r,i)`



Damit ergibt sich als Laufzeit:

$$T(n) = \Theta(n) + T(n/5) + T(7n/10) \in \Theta(n)$$

Frage: Welche Auswirkungen hat dieses Ergebnis auf Quicksort?

Antwort: Man könnte eine verbesserte Wahl des Pivot-Elements implementieren, damit die Laufzeit im worst-case verbessert wird. Dies hat aber praktisch keinen Nutzen, da dies nur bei paarweise verschiedenen Elementen der zu sortierenden Folge nützlich ist. Obiges Verfahren nützt nichts, falls viele gleiche Werte zu sortieren sind, da dann das Aufteilungsverfahren von Quicksort schlecht ist.

Und bei Folgen, die im wesentlichen nur verschiedene Werte enthalten, liefert der randomisierte Quicksort in der Praxis hervorragende Resultate.

5 Graphalgorithmen

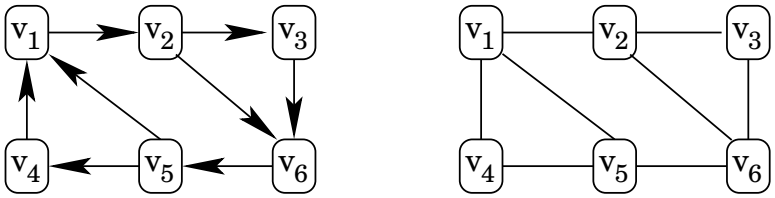
Frage: Wie unterscheidet man gerichtete und ungerichtete Graphen?

Antwort: Ein *gerichteter Graph* $G = (V, E)$ besteht aus

- einer endlichen Menge von *Knoten* $V = \{v_1, \ldots, v_n\}$ und
- einer Menge von gerichteten *Kanten* $E \subseteq V \times V$.

Bei einem *ungerichteten Graphen* $G = (V, E)$ sind die Kanten ungeordnete Paare:

$$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$$



gerichteter Graph ungerichteter Graph

Frage: Was ist ein Weg in einem Graphen?

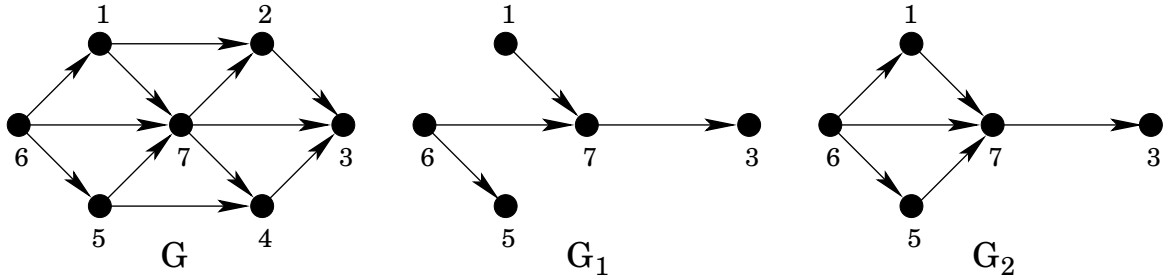
Antwort: Sei $G = (V, E)$ ein gerichteter Graph und seien $u, v \in V$. Dann ist $p = (v_0, v_1, \ldots, v_k)$ ein *gerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.

Der Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.

Oder: Sei $G = (V, E)$ ein ungerichteter Graph und seien $u, v \in V$. $p = (v_0, v_1, \ldots, v_k)$ ist ein *ungerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.

Frage: Was ist der Unterschied zwischen einem Teilgraphen und einem induzierten Teilgraphen?

Antwort: G_1 und G_2 sind Teilgraphen von G. Aber G_1 ist kein induzierter Teilgraph von G, da die Kanten $(6, 1)$ und $(5, 7)$ fehlen. G_2 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G.



- Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- Ein Graph $G' = (V', E')$ heißt *induzierter Teilgraph* von G, falls $V' \subseteq V$ und $E' = E \cap (V' \times V')$ gilt. Der Graph $G|_{V'}$ bezeichnet den durch V' induzierten Teilgraphen von G.

Frage: Welche Aufgabe wollen wir mit der Tiefensuche lösen und wie funktioniert die Tiefensuche auf gerichteten Graphen?

Antwort: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

`tiefensuche(`u`)`:

```
markiere u als „besucht“
dfb[u] := dfbZähler
dfbZähler := dfbZähler + 1
betrachte alle Kanten (u, v) ∈ E:
    falls Knoten v als „unbesucht“ markiert ist:
        tiefensuche(v)
dfe[u] := dfeZähler
dfeZähler := dfeZähler + 1
```

Frage: Welche Kantenarten unterscheiden wir bei der Tiefensuche auf gerichteten Graphen?

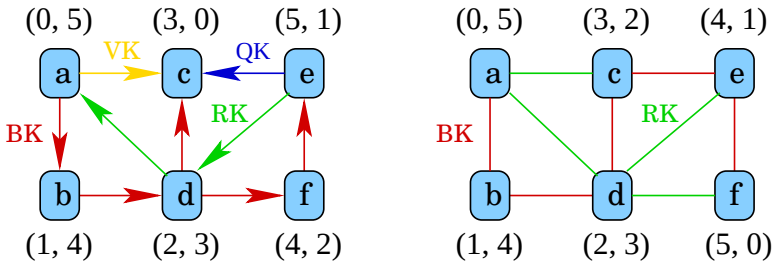
Antwort:

- *Baumkante:* Kante, der die Tiefensuche folgt.
- *Vorwärtskante:* Kante $(u, v) \in E$ mit $\text{dfb}[v] > \text{dfb}[u]$, die aber keine Baumkante ist.
- *Querkante:* Eine Kante $(u, v) \in E$ mit $\text{dfb}[v] < \text{dfb}[u]$ und $\text{dfe}[v] < \text{dfe}[u]$.
- *Rückwärtskante:* Kante $(u, v) \in E$ mit $\text{dfb}[v] < \text{dfb}[u]$ und $\text{dfe}[v] > \text{dfe}[u]$.

Frage: Welche Kantenarten entfallen bei Tiefensuche auf ungerichteten Graphen? Mit Begründung.

Antwort: Es entfallen die Querkanten und die Vorwärtskanten.

- Querkanten werden zu Baumkanten.
- Vorwärtskanten werden zu Rückwärtskanten.



Frage: Wie testen wir einen Graphen auf Kreisfreiheit? Beweisidee?

Antwort: Modifizierte Tiefensuche:

```
markiere alle Knoten als unbesucht
solange ein unbesuchter Knoten v existiert:
    tiefensuche(v)
```

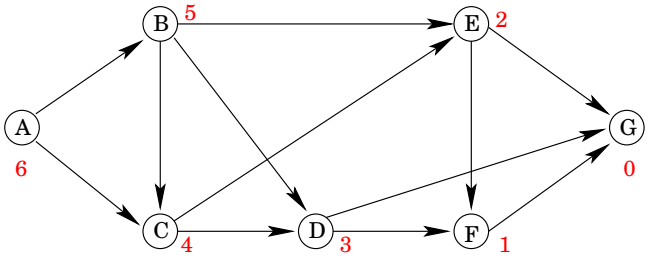
Satz: Ein gerichteter Graph G enthält genau dann einen Kreis, wenn die modifizierte Tiefensuche auf G eine Rückwärtskante liefert.

Frage: Was ist eine topologische Sortierung und wie können wir algorithmisch eine solche bestimmen?

Antwort:

- *Gegeben:* Ein gerichteter Graph $G = (V, E)$.
- *Gesucht:* Eine Nummerierung $\pi(v_1), \ldots, \pi(v_n)$ der Knoten, sodass gilt:

$$(u, v) \in E \;\Rightarrow\; \pi(u) > \pi(v)$$



Algorithmus: Modifizierte Tiefensuche
G ist kreisfrei $\Rightarrow$ dfe-Nummern sind topologische Sortierung!

Frage: Was ist eine starke Zusammenhangskomponente (in gerichteten Graphen) und wie bestimmen wir diese algorithmisch? Mit Beweisidee der Korrektheit.

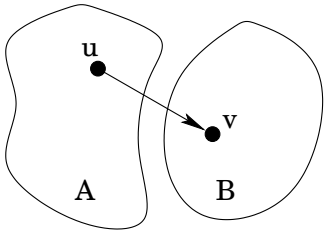
Antwort: Sei $G = (V, E)$ ein gerichteter Graph.

- G ist *stark zusammenhängend*, wenn es zwischen jedem Knotenpaar einen Weg in G gibt. Also: Für alle $u, v \in V$ existiert ein Weg von u nach v und ein Weg von v nach u in G.
- Eine *starke Zusammenhangskomponente* von G ist ein bzgl. der Knotenmenge maximaler, stark zusammenhängender, induzierter Teilgraph von G.

Algorithmus: Gegeben sei ein gerichteter Graph $G = (V, E)$.

1. Modifizierte Tiefensuche auf G mit dfe-Nummerierung der Knoten.
2. Berechne den Graphen $G^R = (V, E^R)$ mit $E^R = \{(v, u) \mid (u, v) \in E\}$, bei dem die Kanten des ursprünglichen Graphen umgedreht werden.
3. Modifizierte Tiefensuche auf G^R, wobei immer mit dem Knoten v mit *größtem dfe-Index aus Schritt 1* gestartet wird.

Satz: Jeder Baum des in Schritt 3 entstandenen Waldes entspricht einer starken Zusammenhangskomponente.



Wenn eine Kante $(u, v) \in E$ existiert, die eine starke Zshg.-komp. A mit einer starken Zshg.-komp. B verbindet,

- dann gilt nach Schritt 1: $\forall w \in E(B) : dfe(u) > dfe(w)$
- dann kann keine Kante von B nach A existieren, da sonst A und B eine einzige starke Zshg.-komp. bilden würden.

Unterscheide zwei Fälle:

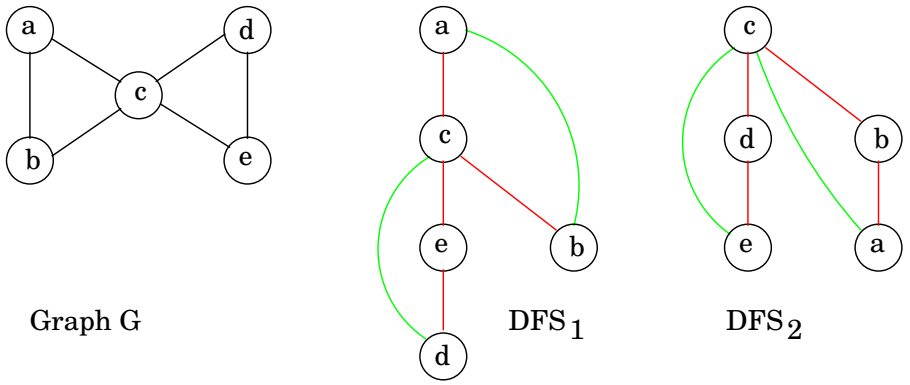
- Falls die Tiefensuche in B beginnt, dann ist die Tiefensuche in B beendet, bevor die Tiefensuche in A startet. $\rightarrow dfe(u) > dfe(v)$
- Falls die Tiefensuche in A beginnt, dann erreicht die Suche irgendwann B, und B wird komplett abgearbeitet, bevor die Suche in A fortgesetzt wird. $\rightarrow dfe(u) > dfe(v)$

Frage: Was bezeichnen wir (in ungerichteten Graphen) mit einem Schnittpunkt, auch Artikulationspunkt genannt, und wie bestimmen wir diese algorithmisch?

Antwort: Sei $G = (V, E)$ ein ungerichteter Graph.

- G heißt *zusammenhängend*, wenn es zwischen jedem Knotenpaar $u, v \in V$ einen Weg in G gibt.
- G heißt *k-fach zusammenhängend*, wenn es zwischen jedem Knotenpaar k knotendisjunkte Wege gibt. Das heißt, außer die Start- und Endknoten sind alle auf den Wegen liegenden Knoten paarweise verschieden.
- Eine *k-fache Zusammenhangskomponente* des Graphen G ist ein maximaler, k-fach zusammenhängender, induzierter Teilgraph von G.
- Ein Knoten u ist ein *Schnittpunkt*, wenn der Graph G ohne Knoten u (und damit auch ohne die zu u inzidenten Kanten) mehr Zusammenhangskomponenten hat als G.

Tiefensuche auf ungerichteten Graphen G:



(a) Es existieren keine Querkanten.

(b) Die Wurzel w eines DFS-Baums ist ein Schnittpunkt, wenn w mehr als einen Teilbaum hat, denn: G ohne w zerfällt wegen Punkt (a) in mehrere Teile, siehe Knoten c bei DFS_2.

(c) Ein innerer Knoten v ist *kein* Schnittpunkt, wenn aus jedem Teilbaum von v eine Rückwärtskante zu einem Vorgänger von v geht, denn: Wenn v aus G entfernt wird, existiert über die Rückwärtskante ein alternativer Weg in den Teilbaum.
Knoten c in DFS_1 ist ein Schnittpunkt: Vom rechten Teilbaum geht zwar eine Rückwärtskante zu einem Vorgänger von c im Baum, aber vom linken Teilbaum nicht.

- Berechne für Knoten v am Ende von `tiefensuche(`v`)`:

$$low(v) := \min\left\{dfb(v), \min_{(v,z)\in RK}\{dfb(z)\}, \min_{(v,y)\in BK}\{low(y)\}\right\}$$

Der low-Wert beschreibt, „wie hoch es im Baum geht über Rückwärtskanten“.

Frage: Was ist ein minimaler Spannbaum?

Antwort: Ein *Baum* ist ein zusammenhängender, kreisfreier Graph. Ein *Spannbaum* $T = (V, E_T)$ eines Graphen $G = (V, E)$ ist ein Baum, der jeden Knoten aus V enthält. Ein solcher Baum besteht aus $|V| - 1$ Kanten. Die *Kosten* eines Spannbaums sind definiert als

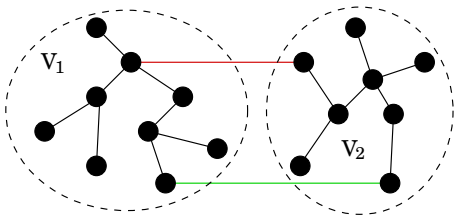
$$c(T) = \sum_{e\in E_T} c(e)$$

Ein minimaler Spannbaum ist ein Spannbaum mit minimalen Kosten unter allen möglichen Spannbäumen des Graphen.

Frage: Korrektheitsbeweise für die Algorithmen zur Berechnung minimaler Spannbäume beruhen auf dem folgenden Satz: Sei (V_1, V_2) eine disjunkte Zerlegung der Knotenmenge V, also $V_1 \cap V_2 = \emptyset$ und $V_1 \cup V_2 = V$. Dann existiert ein minimaler Spannbaum, der die billigste Kante $e = \{u, v\} \in E$ mit $u \in V_1$ und $v \in V_2$ enthält. Skizzieren Sie die Idee des Beweises.

Antwort: Beweis durch Widerspruch:

Wir nehmen an, dass kein minimaler Spannbaum die billigste Kante zwischen V_1 und V_2 enthält. Dann entsteht durch Hinzunahme der **billigsten Kante** ein Kreis.



Durch streichen der teueren Kante zwischen V_1 und V_2 entsteht wieder ein Spannbaum, dessen Gewicht sogar geringer ist als der ursprüngliche Spannbaum. ↯↯

Frage: Was erfolgt die Berechnung eines minimalen Spannbaums nach Prim? Welche Laufzeit hat der Algorithmus?

Antwort: Sei Q eine Datenstruktur (Priority Queue) zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

```
Q := V
key[v] := ∞ for all v ∈ V
key[s] := 0 for some arbitrary s ∈ V
while Q ≠ ∅ do
        u := ExtractMin(Q)
        for all v ∈ Adj(u) do
                if v ∈ Q and c((u,v)) < key[v]
                then  key[v] := c((u,v))
                      π[v] := u
```

Laufzeit:

$$T = \Theta(V) \cdot T_{ExtractMin} + \Theta(E) \cdot T_{DecreaseKey}$$

Frage: Wie wird der minimale Spannbaum nach Kruskal berechnet?

Antwort: Sei $G = (V, E, c)$ ein ungerichteter Graph mit Kostenfunktion $c : E \to \mathbb{R}^+$.

```
A := ∅
for each vertex v ∈ V do
        makeSet(v)
sort the edges of E by non-decreasing weight
for each edge (u,v) ∈ E do
        if findSet(u) ≠ findSet(v)
        then  A := A ∪ {(u,v)}
              union(u, v)
```

Nach Ablauf des Algorithmus enthält die Menge A die Kanten eines minimalen Spannbaums.

Variante: Anstelle des Sortierens können die Kanten auch in eine Priority-Queue Q eingefügt werden, sortiert nach dem Gewicht der Kanten. Die Schleife ändert sich dann zu:

```
while (Q ≠ ∅)
        (u, v) := extractMin(Q)
        ...
```

Frage: Erläutern Sie die Datenstruktur, die wir zur Darstellung der Mengen bei Kruskals Algorithmus verwenden.

Antwort: Insbesondere sollen die Funktionen `findSet` und `union` unterstützt werden. Daher wird die Datenstruktur *Union-Find-Datenstruktur* genannt. Sie unterstützt die Speicherung einer disjunkten Zerlegung einer Menge $S = X_1 \cup X_2 \cup \ldots \cup X_k$ mit $X_i \cap X_j = \emptyset$ für $i \neq j$.

- Speichere jede Klasse X_i in einem Baum.
- Der Repräsentant einer Klasse X_i ist die Wurzel des Baums.
- Die Funktion `findSet(`v`)` liefert den Repräsentanten des Baums, in dem Knoten v gespeichert ist.
- Damit ein Knoten schnell gefunden werden kann, werden Zeiger auf alle Elemente $v \in S$ gespeichert.
- Die Funktion `union` hängt den kleineren (bzw. flacheren) Baum an die Wurzel des größeren (bzw. tieferen) an.

Frage: Welche Technik nutzen wir, um die Operationen bei der Union-Find-Datenstruktur zu beschleunigen? Welche Laufzeit ergibt sich dann für Kruskals Algorithmus?

Antwort: Die Kosten einer `findSet`-Operation sind abhängig von der Höhe der Bäume.

- Es wäre günstig, alle Knoten direkt an die Wurzel zu hängen. Das aber würde die `union`-Operation teurer machen als bisher.
- Stattdessen: Verkürze während der `findSet`-Operation die Pfadlängen. Die Pfadkomprimierung macht die `findSet`-Methode ungefähr doppelt so teuer wie vorher, die asymptotische Laufzeit bleibt gleich.
- Eine amortisierte Laufzeitanalyse liefert: Kruskals Algorithmus hat für alle praktischen Eingaben eine lineare Laufzeit.

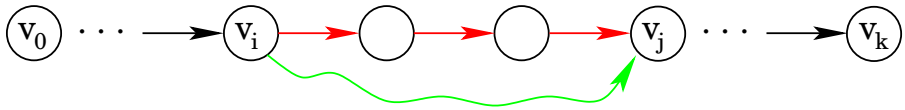
Frage: Wie funktioniert die kürzeste Wegeberechnung nach Dijkstra? Welche Laufzeit hat der Algorithmus?

Antwort: Es werden alle kürzesten Wege von einem Startknoten aus gesucht, (single source shortest paths). Sei Q eine Datenstruktur (Priority Queue) zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

```
Q := V
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
while Q ≠ ∅ do
        u := ExtractMin(Q)
        for all v ∈ Adj(u) do
                if v ∈ Q and d[v] > d[u] + c((u, v))
                then  d[v] := d[u] + c((u, v))
                      π[v] := u
```

Frage: Bei Dijkstras Algorithmus wird das Optimalitätsprinzip genutzt. Erklären Sie, was Optimalitätsprinzip in diesem Fall heißt und warum es gilt.

Antwort: Jeder Teilweg eines kürzesten Weges ist ebenfalls ein kürzester Weg. Betrachten wir einen kürzesten Weg $(v_0, v_1, v_2, \ldots, v_k)$ von v_0 nach v_k. Dann ist jeder Teilweg von v_i nach v_j für $i < j$ auch ein kürzester Weg von v_i nach v_j.

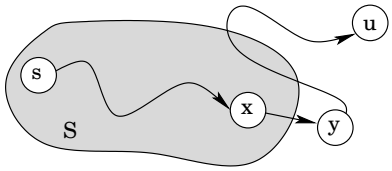


Cut and paste: Gäbe es einen kürzeren Weg von v_i nach v_j, dann könnte auch der Weg von v_0 nach v_k verkürzt werden.

Frage: Skizzieren Sie die Idee des Korrektheitsbeweises zu Dijkstras Algorithmus.

Antwort: Wir zeigen, dass $d[v] = \delta(s, v)$ gilt, wenn v zur Menge S hinzu genommen wird. Da die Werte von $d[v]$ nur kleiner werden können, ist damit die Aussage gezeigt.

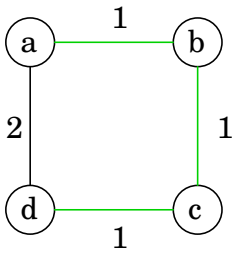
Angenommen, u ist der erste Knoten, der zu S hinzu genommen wird, für den $d[u] \neq \delta(s, u)$ gilt. Sei y der erste Knoten aus $V - S$ auf einem kürzesten Weg von s nach u, und sei x der Vorgänger von y auf diesem Weg.



- Es gilt $d[x] = \delta(s, x)$, da u der erste Knoten ist, der die Invariante verletzt.
- Da Teilwege von kürzesten Wegen auch kürzeste Wege sind, wurde $d[y]$ auf $\delta(s, x) + c((x, y)) = \delta(s, y)$ gesetzt, als die Kante (x, y) nach Hinzunahme von x zu S untersucht wurde.
  Also gilt $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$, denn $c : E \to \mathbb{R}^+$. Hier geht die Vorbedingung ein, dass die Kanten keine negativen Kosten haben!
- Aber es gilt $d[u] \leq d[y]$, weil der Algorithmus u wählt.
- Also: $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ ↯↯

Frage: Warum benötigen wir einen neuen Algorithmus zur Berechnung kürzester Wege? Sind die kürzesten Wege nicht bereits durch einen minimalen Spannbaum gegeben?

Antwort: Betrachten wir dazu ein Beispiel.



- Der minimale Spannbaum im nebenstehenden Graphen ist eindeutig und durch die grün markierten Kanten gekennzeichnet.
- Der kürzeste Weg von a nach d ist allerdings durch die Kante $\{a, d\}$ gegeben, die nicht zum minimalen Spannbaum gehört.

Frage: Kürzeste Wege bei Graphen mit negativen Kantengewichten können mittels des Algorithmus von Bellman/Ford berechnet werden. Wie funktioniert dieser Algorithmus?

Antwort: single source shortest paths

```
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
for i := 1 to |V| − 1 do
        for each edge (u, v) ∈ E do
                if d[v] > d[u] + c((u, v))
                then  d[v] := d[u] + c((u, v))
for each edge (u, v) ∈ E do
        if d[v] > d[u] + c((u, v))
        then  report: „negative-weight cycle exists“
```

Frage: Welche Laufzeit hat der Algorithmus von Bellman/Ford? Mit Erklärung!

Antwort:

- Die äußere Schleife wird $(\mathcal{V} - 1)$-mal durchlaufen.
- Die innere Schleife wird $\mathcal{E}$-mal durchlaufen.
- Alle Operationen der inneren Schleife kosten Zeit $\mathcal{O}(1)$.

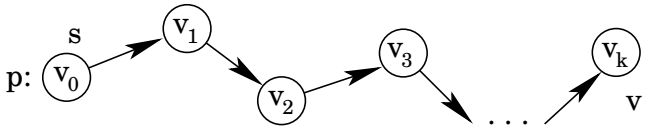
$\rightarrow$ Gesamte Laufzeit in $\Theta(\mathcal{V} \cdot \mathcal{E})$.

Frage: Skizzieren Sie die Idee zur Korrektheit des Algorithmus von Bellman/Ford.

Antwort: Ein kürzester Weg, der keine negativen Kreise enthält, besucht keinen Knoten zweimal. Daher besteht ein solcher Weg aus höchstens $\mathcal{V} - 1$ Kanten.

Der Graph G enthalte keine negativen Kreise.

- Sei $v \in V$ ein beliebiger Knoten, und betrachte einen kürzesten Weg p von s nach v mit minimaler Anzahl Kanten.



  Da p kürzester Weg ist: $\delta(s, v_i) = \delta(s, v_{i-1}) + c((v_{i-1}, v_i))$.
- Initial gilt: $d[v_0] = 0 = \delta(s, v_0)$
- Wir betrachten die Durchläufe der äußeren Schleife:
  - nach 1. Durchlauf durch E gilt: $d[v_1] = \delta(s, v_1)$.
  - nach 2. Durchlauf durch E gilt: $d[v_2] = \delta(s, v_2)$.

  $\vdots$
  - nach k. Durchlauf durch E gilt: $d[v_k] = \delta(s, v_k)$.
- Ein kürzester Weg besteht aus höchstens $\mathcal{V} - 1$ Kanten.

Frage: Das all-pairs-shortest-path-Problem kann mittels des Floyd/Warshall-Algorithmus berechnet werden. Beschreiben Sie den Algorithmus.

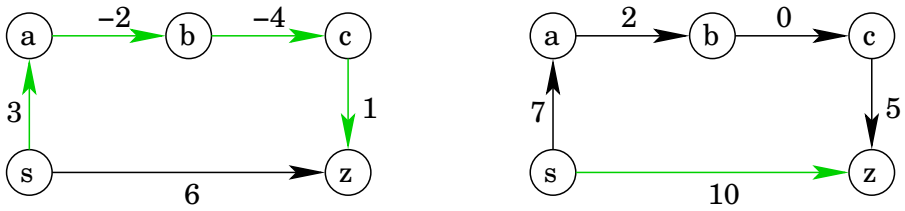
Antwort: Mittels dynamische Programmierung all-pairs-shortest-path berechnen.

Sei d_{ij}^k die Länge eines kürzesten Weges von i nach j, der nur über Knoten mit Nummern kleiner gleich k läuft. Dann gilt:

- $d_{ij}^0 = c(i, j)$
- $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$

Frage: Warum addieren wir nicht einfach den Betrag des kleinsten Kantengewichts auf alle Kantengewichte auf und berechnen dann mittels Dijkstras Algorithmus die kürzesten Wege?

Antwort: Weil es nicht funktioniert, wie folgendes Beispiel zeigt.



- Der kürzeste Weg zwischen Startknoten s und Zielknoten z ist jeweils grün markiert.
- Links ist der Originalgraph zu sehen, im rechten Graphen wurde auf jedes Kantengewicht der Wert 4 addiert.

Frage: Wie ist das Problem *maximaler Fluss* in einem Netzwerk definiert?

Antwort:

- Gegeben:
  - Ein gewichteter Graph $G = (V, E, c)$ mit einer Kostenfunktion $c : E \to \mathbb{Q}^+$.
  - Eine Quelle $s \in V$ und eine Senke $t \in V$, wobei $s \neq t$ gilt.
- Gesucht: Maximaler Fluss im Netzwerk von Quelle s nach Senke t.

Eine *Flussfunktion* f eines Netzwerks $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist eine Funktion $f : E \to \mathbb{Q}^+$ mit:

- *Kapazitätsbeschränkung:* $\forall e \in E : \ 0 \leq f(e) \leq c(e)$
- *Flusserhaltung:* $\forall v \in V - \{s, t\}$:

$$\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$$

Der *Fluss* $F(f)$ im Netzwerk $H = (V, E, c, \mathbb{Q}^+, s, t)$ ist definiert als

$$F(f) = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e)$$

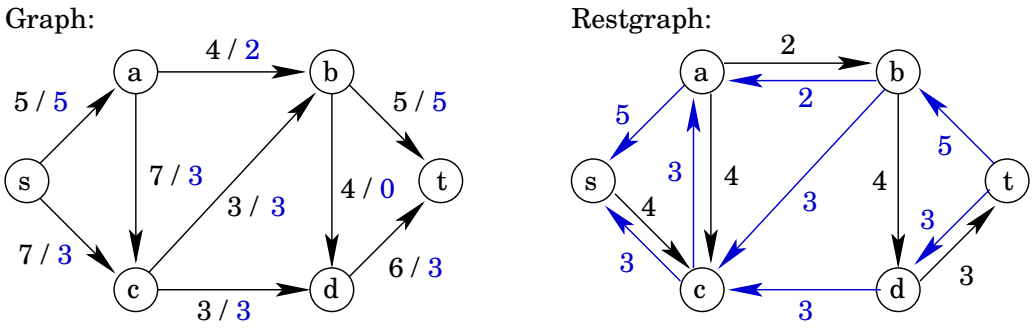
oder analog:

$$F(f) = \sum_{e \in Out(s)} f(e) - \sum_{e \in In(s)} f(e)$$

Frage: Was ist ein Restgraph und wofür wird er bei Fluss-Algorithmen benötigt?

Antwort: Zu einem Graphen G und einer Flussfunktion f enthält der Restgraph alle vergrößerbaren Pfade von s nach t im Graphen G. Für jede Kante $e \in E$ von u nach v gibt es im Restgraphen

- eine (Vorwärts-)Kante von u nach v mit Gewicht $c(e) - f(e)$, falls $c(e) > f(e)$,
- und eine (Rückwärts-)Kante von v nach u mit Gewicht $f(e)$, falls $f(e) > 0$.



Jeder Weg von s nach t im Restgraphen $G_R(f)$ ist ein zunehmender Pfad in G.

Frage: Maximaler Fluss nach Ford/Folkersson? Laufzeit? Probleme?

Antwort: Starte mit beliebiger Flussfunktion f, in der Regel $f(e) := 0 \; \forall e \in E$. Suche einen beliebigen zunehmenden Pfad P (z.B. mittels Tiefensuche) im Restgraphen $G_R(f)$ und erhöhe den Fluss entlang des Pfades um $\Delta(P)$, dabei ist $\Delta(P)$ das kleinste Kantengewicht über alle Kanten e im Pfad P. Wiederhole diesen Schritt, solange ein Pfad von s nach t im Restgraphen existiert.

Bemerkung:

- Der Algorithmus terminiert nicht immer für irrationale Kapazitäten.
- Der Algorithmus konvergiert für irrationale Kapazitäten nicht unbedingt gegen $F_{\max}$.
- Für ganzzahlige Kapazitäten ist die Anzahl der Flussvergrößerungen durch $F_{\max}$ beschränkt und daher nicht nur abhängig von der Größe des Graphen sondern auch vom maximalen Kantengewicht!

Frage: Maximaler Fluss nach Edmond/Karp? Idee? Laufzeit?

Antwort: Wähle immer einen zunehmenden Pfad mit der minimalen Anzahl Kanten.

Werden immer nur Pfade mit minimaler Anzahl von Kanten zur Flussvergrößerung herangezogen (kürzeste Pfade), so erhöht sich die Anzahl der Kanten auf den kürzesten Pfaden nach höchstens $\mathcal{E}$ Iterationen mindestens um 1. $\rightarrow$ Die Pfadlänge ist monoton steigend.

Laufzeit:

- Anzahl Iterationen: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$, da die Pfadlänge monoton steigend ist, und jeder Pfad aus höchstens $\mathcal{V}$ Knoten besteht.
- Kürzeste Pfade finden: Breitensuche in Zeit $\mathcal{O}(\mathcal{E})$.
- Gesamte Laufzeit: $\mathcal{O}(\mathcal{E}^2 \cdot \mathcal{V})$

Frage: Maximaler Fluss nach Dinic? Idee? Laufzeit?

Antwort: Finde alle kürzesten Wege mit gleicher Anzahl von Kanten in einer Runde.

1. Bilde den Schichtengraphen $L(G_R(f))$. Dieser kann mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{E})$ aufgebaut werden.
2. Bestimme eine Flussfunktion f' für den Schichtengraphen $L(G_R(f))$, die nicht durch einen zunehmenden Pfad von s nach t erhöhbar ist und addiere f' zu f hinzu.
   Mit anderen Worten: Jeder Weg von s nach t im Schichtengraphen enthält eine gesättigte Kante. Wir nennen f' einen blockierenden Fluss für den Schichtengraphen.
3. Wiederhole Schritt 1 und 2 solange, bis der Fluss f maximal ist, also bis es keinen Weg von s nach t im Schichtengraphen $L(G_R(f))$ gibt.

Wie findet man einen blockierenden Fluss? Mittels wiederholter Tiefensuche von Knoten s aus. Wir modifizieren die Tiefensuche so, dass die Suche endet, wenn der Knoten t erreicht wurde, oder wenn ein Knoten ohne auslaufende Kanten erreicht wurde. $\rightarrow$ Laufzeit $\mathcal{O}(\mathcal{V})$

- Wenn der Knoten t durch die Tiefensuche erreicht wird, wurde ein zunehmender Pfad P von s nach t gefunden:
  - Verkleinere die Kapazitäten der Kanten auf P um $\Delta(P)$.
  - Mindestens eine Kante erhält die Kapazität 0: Entferne diese Kante aus dem Schichtengraph. $\rightarrow$ Laufzeit: $\mathcal{O}(\mathcal{V})$
- Wenn Knoten t nicht erreicht wird, entferne die letzte Kante des Weges, damit keine Endlosschleifen auftreten.

Da in jedem Durchlauf mindestens eine Kante aus dem Schichtengraphen entfernt wird, gibt es höchstens $\mathcal{O}(\mathcal{E})$ Iterationen. Als Laufzeit zur Bestimmung eines blockierenden Flusses erhalten wir insgesamt also $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$.

Laufzeit insgesamt:

- Nach jeder Iteration der Schritte 1 und 2 des Algorithmus erhöht sich die Anzahl der Kanten in den kürzesten Wegen um mindestens 1. Da ein kürzester Weg keinen Knoten mehrfach besucht, gibt es höchstens $\mathcal{O}(\mathcal{V})$ Iterationen.
- Bestimmen des blockierenden Flusses in einer solchen Iteration kostet $\mathcal{O}(\mathcal{E} \cdot \mathcal{V})$ Zeit.
- Insgesamt ergibt sich also als Laufzeit: $\mathcal{O}(\mathcal{E} \cdot \mathcal{V}^2)$

Frage: Was ist der wesentliche Unterschied zwischen dem Push-Relabel-Algorithmus und den anderen Flussalgorithmen, die wir kennengelernt haben?

Antwort: Alle anderen Flussalgorithmen gehen von einer Flussfunktion aus und versuchen diese so zu verbessern, dass am Ende der Wert der Flussfunktion maximal ist. Der Push/Relabel-Algorithmus arbeitet mit einem Präfluss: Die Kapazitätsbeschränkungen werden eingehalten, aber es kann mehr in einen Knoten hinein fließen, als wieder heraus fließt.

Frage: Wie funktioniert der Push-Relabel-Algorithmus?

Antwort: Nach der Initialisierung werden solange Push- oder Relabel-Operationen ausgeführt, bis keine aktiven Knoten (solche mit Überschuss) mehr existieren. Überschuss kann nur von einem höheren zu einem niedrigeren Knoten abgebaut werden.

while $\exists$ active node $v \in V - \{s, t\}$ **do**
  if $\exists$ admissible edge $(v, w) \in E_f$
    then PUSH(v,w)
  else RELABEL(v)

PUSH(v,w)
  if $(v, w) \in E$ **then**
    $f(v, w) := f(v, w) + \min\{excess_f(v), c_f(v, w)\}$
  else $f(v, w) := f(v, w) - \min\{excess_f(v), c_f(v, w)\}$

RELABEL(v)
  $height(v) := \min\{height(w) + 1 \mid (v, w) \in E_f\}$

Frage: Es war recht aufwändig zu zeigen, dass der Push-Relabel-Algorithmus terminiert. Was waren die wesentlichen zwei Ideen in dem Beweis?

Antwort:

- Ein Überschuss an einem beliebigen Knoten kann immer zur Quelle hin wieder abgebaut werden.
- Die Knoten können nicht beliebig angehoben werden.

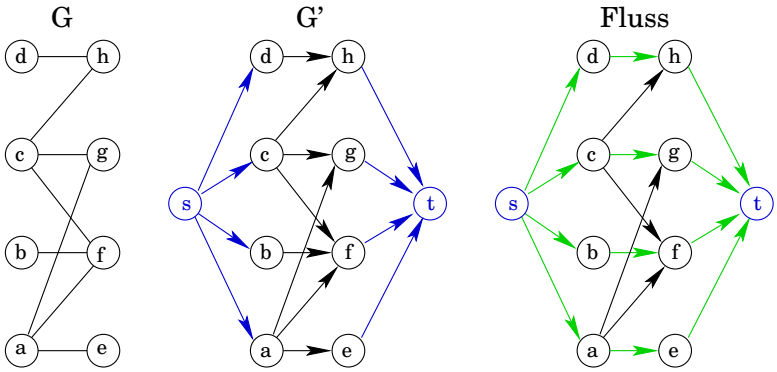
Frage: Wie kann ein Maximum-Matching in bipartiten Graphen berechnet werden?

Antwort: Mittels Flussalgorithmus. Konstruiere zu dem gegebenen Graphen $G = (V, E)$ das Netzwerk $G' = (V', E', c)$ mit

- $V' = V \cup \{s, t\}$
- $E' = \{s\} \times V_1 \cup \{(u, v) \mid \{u, v\} \in E, u \in V_1, v \in V_2\} \cup V_2 \times \{t\}$
- $c : E' \to \mathbb{N}$ mit $c(e) = 1$ für alle Kanten $e \in E'$.

Algorithmus

- Bestimme für G' einen max. Fluss durch zunehmende Pfade P mit $\Delta(P) = 1$. Dann hat jede Kante in G' den Flusswert 0 oder 1.
- Die Kanten $\{u, v\} \in E$ mit $f((u, v)) = 1$ bilden ein Maximum-Matching in G.

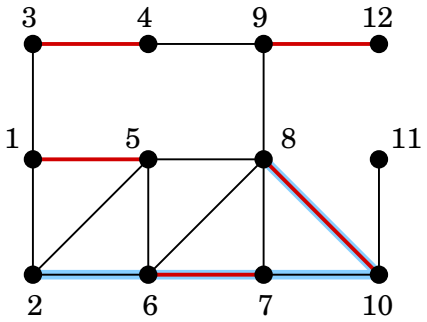


Frage: Wie kann ein Maximum-Matching ohne Flussalgorithmen in bipartiten Graphen berechnet werden?

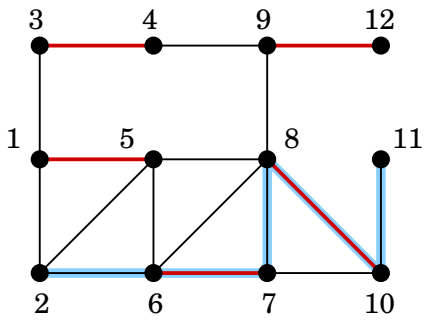
Antwort: Wir nutzen zunehmend alternierende Wege, um ein bestehendes Matching zu vergrößern. Ein Weg $P = (v_1, \ldots, v_k)$, der abwechselnd aus freien und gebundenen Kanten besteht, heißt *alternierender Weg*. Ein solcher Weg heißt *zunehmend alternierender Weg*, wenn $v_1 \neq v_k$ gilt und beide Endknoten v_1 und v_k mit keiner gebundenen Kante inzident sind. Um ein Matching M zu vergrößern, entfernen wir alle gebundenen Kanten von P aus M und fügen alle freien Kanten von P zu M hinzu.

Frage: Warum kann diese Idee nicht einfach auf beliebige Graphen angewendet werden?

Antwort: In bipartiten Graphen können zunehmend alternierende Wege mittels Breitensuche in Zeit $\mathcal{O}(\mathcal{V} + \mathcal{E})$ berechnet werden. Auf allgemeinen Graphen funktioniert das nicht so einfach. Es kann vorkommen, dass ein Knoten auf einem Weg gerader Länge und auf einem anderen Weg ungerader Länge erreichbar ist. Bei ungünstiger Wegwahl kann der gefundene Weg nicht zu einem zunehmend alternierenden Weg erweitert werden.



Der Weg $P_1 = (2, 6, 7, 10, 8)$ hat die Länge 4, also gerade Länge.



Der Weg $P_2 = (2, 6, 7, 8)$ hat die Länge 3, also ungerade Länge.

Ein Kreis ungerader Länge heißt *Blüte*. Wird bei der Suche nach einem zunehmend alternierenden Weg eine Blüte gefunden, so schrumpfen wir die Blüte auf einen Knoten zusammen: Jede Kante, die zuvor mit einem Knoten der Blüte verbunden war, führt jetzt zu diesem neuen Knoten.

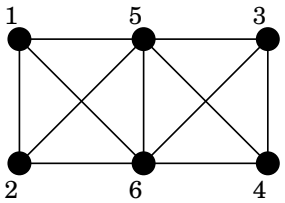
6 Spezielle Graphklassen

Frage: Wann heißt eine Grapheigenschaft monoton, wann hereditär?

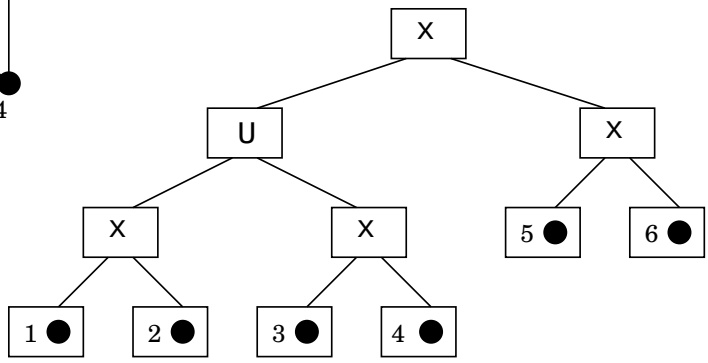
Antwort: Eine Grapheigenschaft heißt *monoton*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner Subgraphen diese Eigenschaft hat.

Eine Grapheigenschaft heißt *hereditär*, falls für jeden Graphen mit dieser Eigenschaft gilt, dass auch jeder seiner induzierten Subgraphen diese Eigenschaft hat.

Frage: Für zwei Co-Graphen G_1 und G_2 sind auch $G_1 \cup G_2$ und $G_1 \times G_2$ Co-Graphen. Geben Sie zu dem folgenden Co-Graphen den Co-Baum an.



Antwort:



Frage: Wie kann mit Hilfe eines Co-Baums das Independent-Set-Problem für Co-Graphen effizient gelöst werden?

Antwort: Sei T der Co-Baum zum gegebenen Co-Graphen G.

- Für jedes Blatt v in T setze $\alpha(G[v]) = 1$.
- Für jeden mit $\cup$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \alpha(G[v_1]) + \alpha(G[v_2])$
- Für jeden mit $\times$ gekennzeichneten Knoten v mit den Kindern v_1 und v_2 in T setze $\alpha(G[v]) = \max\{\alpha(G[v_1]), \alpha(G[v_2])\}$

Frage: Wann nennt man einen Graphen chordal? Ist Chordalität eine monotone oder hereditäre Grapheigenschaft? Begründen Sie Ihre Antworten.

Antwort: Ein Graph heißt *chordal*, falls er keinen induzierten Kreis C_k mit $k \geq 4$ enthält. In anderen Worten: Jeder Kreis der Länge mindestens vier besitzt eine Sehne. Eine *Sehne* ist eine Kante zwischen zwei auf dem Kreis nicht benachbarten Knoten.

Die Eigenschaft *chordal* ist nicht monoton. Wenn man eine Sehne entfernt, zerstört man die Eigenschaft.

Die Eigenschaft *chordal* ist hereditär. Annahme: Durch entfernen von Knoten entsteht ein Kreis der Länge größer gleich vier, der keine Sehne hat. Durch hinzufügen des Knotens und der inzidenten Kanten entsteht keine Sehne in dem Kreis. Also ist schon der ursprüngliche Graph nicht chordal. $\lightning$

Frage: Wie kann bei chordalen Graphen mit Hilfe des perfekten Knoten-Eliminationsschemas das Maximum-Clique-Problem effizient gelöst werden?

Antwort: Sei σ ein PES.

$K := \emptyset$
for all node u in order of σ **do**
  compute $X_u := \{v \in V \mid \{u, v\} \in E, \sigma(v) > \sigma(u)\}$
  if $|K| < |\{u\} \cup X_u|$ **then**
    $K := \{u\} \cup X_u$
output K

Alle zu u benachbarten Knoten v mit $\sigma(v) > \sigma(u)$ bilden eine Clique, sonst wäre σ kein PES.

Frage: Wie sind Vergleichbarkeitsgraphen (comparability graphs) definiert?

Antwort: Ein ungerichteter Graph heißt *Vergleichbarkeitsgraph*, wenn eine Orientierung der Kanten existiert, so dass der orientierte Graph transitiv ist.

Frage: Wie kann bei Vergleichbarkeitsgraphen das Maximum-Clique-Problem effizient gelöst werden?

Antwort: Bestimme einen längsten Weg in $G = (V, E)$, denn alle Knoten auf einem Weg stellen eine Clique dar.

- berechne eine transitive Orientierung $G' = (V, E')$
- berechne eine topologische Sortierung zu G'
- induktiv: $dist(v_1) := 0$, dann

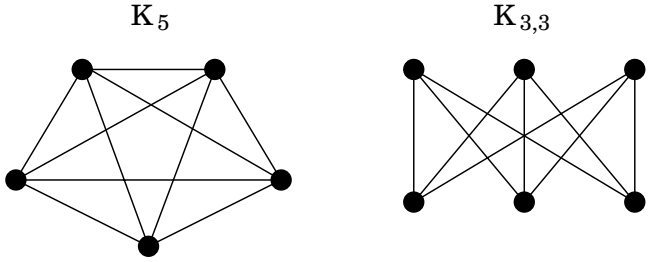
$$dist(v_i) := \max_{\substack{j=1,\ldots,i-1 \\ (v_j, v_i) \in E'}} \{dist(v_j) + 1\}$$

Frage: Wann bezeichnet man einen Graphen als planar?

Antwort: Ein Graph, der kreuzungsfrei in der Ebene gezeichnet werden kann, heißt planar.

Frage: Welche zwei Graphen spielen beim Planaritätstest eine wichtige Rolle? Mit Zeichnung.

Antwort: Der vollständige Graph K_5 und der vollständig bipartite Graph $K_{3,3}$ sind nicht planar. Auch die Unterteilungsgraphen dieser Graphen sind nicht planar.



Frage: Wie kann man zeigen, dass der Graph K_5 nicht planar ist?

Antwort: Nach Eulers Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Da jede Kante höchstens 2 Flächen begrenzt und jede Fläche durch mindestens 3 Kanten begrenzt ist, gilt für eine planare Einbettung eines Graphen $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$. Der K_5 hat 10 Kanten, nach der Formel dürften es aber nur 9 Kanten sein.

Frage: Wie kann man zeigen, dass der Graph $K_{3,3}$ nicht planar ist?

Antwort: Nach Eulers Polyeder-Formel gilt $\mathcal{V} - \mathcal{E} + f = 2$. Da jede Kante höchstens 2 Flächen begrenzt und jede Fläche durch mindestens 4 Kanten begrenzt ist, gilt für eine planare Einbettung eines bipartiten Graphen $\mathcal{E} \leq 2 \cdot \mathcal{V} - 4$. Der $K_{3,3}$ hat 9 Kanten, nach der Formel dürften es aber nur 8 Kanten sein.

Frage: Wie kann man zeigen, dass jeder planare Graph mit höchstens sechs Farben gefärbt werden kann?

Antwort: Es gibt einen Knoten v mit Knotengrad höchstens 5. Hätte jeder Knoten einen Knotengrad von mindestens 6, dann gäbe es mindestens $\frac{6 \cdot \mathcal{V}}{2} = 3 \cdot \mathcal{V}$ viele Kanten in dem Graphen, was aber nicht sein kann, da jeder planare Graph höchstens $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$ viele Kanten hat.

- Wir berechnen eine Färbung für $G - \{v\}$ mit 6 Farben.
- Anschließend fügen wir v mit den dazu gehörenden Kanten wieder ein.

Da der Knoten v in G höchstens 5 Nachbarn hat, ist noch eine Farbe für v frei.

7 Vorrangwarteschlangen

Frage: Für welche Algorithmen benötigen wir Vorrangwarteschlangen?

Antwort: Kürzeste Wege nach Dijkstra, Minimaler Spannbaum nach Prim. Bei Kruskals Algorithmus zur Berechnung minimaler Spannbäume kann anstelle des Sortierens der Kanten zu Beginn auch eine Priority-Queue verwendet werden. (In WSY haben wir den $A^\star$-Algorithmus kennengelernt.)

Frage: Welche Operationen soll eine solche Priority-Queue unterstützen?

Antwort:

- `MakeHeap()` erzeugt neuen Heap ohne Elemente
- `Insert(`H, x`)` fügt Knoten x in Heap H ein

- `Minimum(`H`)` liefert Zeiger auf Knoten mit minimalem Element im Heap H
- `ExtractMin(`H`)` entfernt minimales Element aus Heap H und liefert Zeiger auf den Knoten
- `Union(`H_1, H_2`)` erzeugt neuen Heap, der die Elemente aus H_1 und H_2 enthält
- `DecreaseKey(`H, x, k`)` weist Knoten x im Heap H neuen, kleineren Wert k zu (Zeiger auf x muss bekannt sein)
- `Delete(`H, x`)` entfernt Knoten x aus Heap H (Zeiger auf x muss bekannt sein)

Frage: Wir können eine solche Priority-Queue mit einer verketteten Liste bzw. mit einem Binär-Heap implementieren. Welche Laufzeiten haben dann die einzelnen Operationen?

Antwort:

	Linked List	Binary Heap
`MakeHeap`	$\Theta(1)$	$\Theta(1)$
`Insert`	$\Theta(1)$	$\Theta(\log n)$
`Minimum`	$\Theta(n)$	$\Theta(1)$
`ExtractMin`	$\Theta(n)$	$\Theta(\log n)$
`Union`	$\Theta(1)$	$\Theta(n)$
`DecreaseKey`	$\Theta(1)$	$\Theta(\log n)$
`Delete`	$\Theta(1)$	$\Theta(\log n)$

Frage: Wir hatten Linksbäume als Implementierung einer Vorrangwarteschlange kennengelernt. Welche Struktur und welche Eigenschaften hat ein solcher Linksbaum?

Antwort: Linksbäume (leftist trees) sind binäre heap-geordnete Bäume, die Elemente mit Schlüsselwerten speichern. Die Schlüsselwerte der Kinder sind größer als der Schlüsselwert der Eltern (Min-Heap). Jeder Knoten besitzt einen Distanzwert:

- die Blätter haben den Distanzwert 0
- innere Knoten haben den Distanzwert des rechten Kindes plus 1
- Distanzwert rechtes Kind $\leq$ Distanzwert linkes Kind

$\rightarrow$ das rechteste Blatt hat eine Tiefe von $\mathcal{O}(\log(n))$

Frage: Die wichtigste Operation ist das Verschmelzen zweier Linksbäume. Wie funktioniert das?

Antwort: $\textsc{Union}(D_1, D_2)$ wird rekursiv definiert:

- wenn D_1 oder D_2 ein Blatt ist, dann ist das Ergebnis der Linksbaum D_2 bzw. D_1
- o.B.d.A: der Schlüssel an Wurzel von D_1 ist kleiner als der Schlüssel an Wurzel von D_2 – sonst tausche Linksbäume
  (a) $\textsc{Union}(D_1.\textsc{rechts}, D_2)$
  (b) ist die Distanz vom (neuen) rechten Teilbaum von D_1 größer als die Distanz vom linken Teilbaum von D_1, dann werden die Teilbäume von D_1 vertauscht

---

Frage: Wie werden bei Linksbäumen die anderen Operationen auf die Verschmelze-Operation zurückgeführt und welche Laufzeiten ergeben sich?

Antwort:

- $\textsc{Insert}(D, x)$: Erzeuge einen Linksbaum E, der nur einen einzigen inneren Knoten x mit Distanz 1 hat. Führe dann $\textsc{Union}(D, E)$ aus. $\rightarrow$ Laufzeit: $\mathcal{O}(\log(n))$

- ExtractMin(D): Entferne die Wurzel und verschmelze die beiden Teilbäume der Wurzel. → Laufzeit: $\mathcal{O}(\log(n))$
- Delete(D, x):
  - der Linksbaum zerfällt beim Entfernen von x in den oberhalb von x liegenden Teilbaum und in den linken und rechten Teilbaum
  - ersetze den Knoten x durch ein Blatt b
  - tausche ggf. b mit seinem Geschwister, um links den Teilbaum mit größerer Distanz anzuordnen
  - adjustiere die Distanzwerte von b bis zur Wurzel
  - verschmelze die drei Teilbäume miteinander

  → Laufzeit: $\mathcal{O}(\log(n))$ Obwohl das Adjustieren der Distanzen und Vertauschen von Teilbäumen auf dem Pfad zur Wurzel proportional zur Höhe des Baumes $\Theta(n)$ ist.
- DecreaseKey(D, x, k): Lösche Knoten x aus D mittels Delete(D, x). Ändere den Schlüssel von x auf k und füge ihn mittels Insert(D, x) ein. → Laufzeit: $\mathcal{O}(\log(n))$

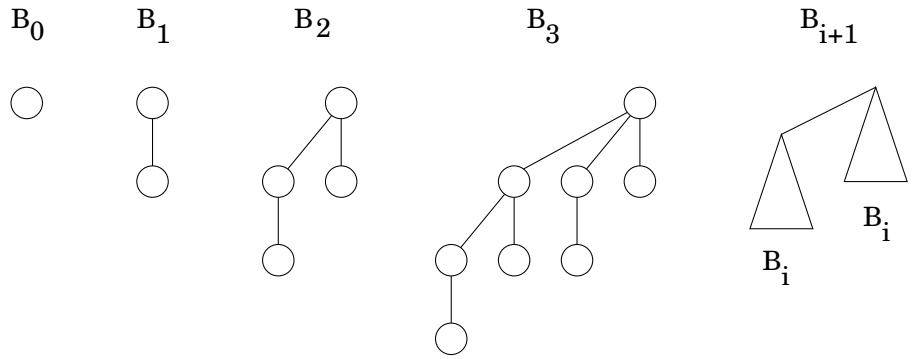
Frage: Wir hatten Binomial-Queues als Implementierung von Vorrangwarteschlangen kennengelernt. Welche Struktur und welche Eigenschaften hat eine solche Binomial-Queue?

Antwort: Binomial-Queues setzen sich aus einzelnen Binomialbäumen zusammen. Dabei sind Binomialbäume heap-geordnete Bäume, die in allen Knoten Elemente mit Schlüsselwerten speichern → (Min-Heap).

Ein Binomialbaum vom Typ

- B_0 besteht aus genau einem Knoten.
- B_{i+1}, $i \geq 0$, besteht aus zwei Kopien der Binomialbäume vom Typ B_i, indem man die Wurzel der einen Kopie zum Kind der Wurzel der anderen Kopie macht.

Ihren Namen verdanken die Binomial-Queues B_k der Tatsache, dass auf der Ebene i die Anzahl der Knoten gerade $\binom{k}{i}$ beträgt



wichtige Eigenschaften eines Binomialbaums vom Typ B_i:

- 2^i Knoten und Höhe $i + 1$
- Teilbäume der Wurzel sind vom Typ $B_{i-1}, B_{i-2}, \ldots, B_0$

Speichern von n Elementen: Anzahl Binomialbäume = Anzahl Einsen in Binärdarstellung von $n = (d_{m-1} \ldots d_0)$

$$\text{Baum vom Typ } B_j \text{ wird benötigt} \iff d_j = 1$$

Beispiel: $n = 13 = (1101)_2 \rightarrow$ Bäume vom Typ B_3, B_2 und B_0

Binomial-Heap/Queue vom Typ D_n: Repräsentation einer Menge mit n Elementen durch Binomialbäume, wobei die Wurzeln der Bäume in einer verketteten Liste gespeichert sind. Die Anzahl der Bäume ist höchstens $\log(n)$.

Frage: Wichtigste Operation bei den Binomial-Queues ist auch wieder die Union-Operation. Wie funktioniert das Verschmelzen zweier Binomial-Queues und welche Laufzeit ergibt sich?

Antwort: Union(H_1, H_2)

- zwei Bäume vom Typ $B_0 \rightarrow$ Baum vom Typ B_1
- zwei Bäume vom Typ $B_1 \rightarrow$ Baum vom Typ B_2
- usw. analog zur Addition zweier Dualzahlen

$\rightarrow$ Laufzeit: $\Theta(\log(n+m))$

Frage: Wie werden bei Binomial-Queues die anderen Operationen auf die Union-Operation zurückgeführt und welche Laufzeiten ergeben sich?

Antwort:

- `Insert(`H, x`)`: Vereinige H und den Baum vom Typ B_0, der nur den Knoten x enthält. $\rightarrow$ Laufzeit: $\Theta(\log(n))$
- `ExtractMin(`H`)`: Das Entfernen des Knotens, der das Minimum enthält, führt dazu, das Teilbäume (wieder Binomial-Bäume) entstehen, die mit den anderen Bäumen vereinigt werden. $\rightarrow$ Laufzeit: $\Theta(\log(n))$
- `DecreaseKey(`H, x, k`)`: Setze den Schlüssel von Knoten x auf den Wert k und führe dann `UpHeap(`H, x`)` aus: Laufe im Baum hoch und vertausche den Knoten mit seinem Vorgänger, falls der Vorgänger einen größeren Wert speichert. $\rightarrow$ Laufzeit: $\Theta(\log(n))$
- `Delete(`H, x`)`: Führe zunächst `DecreaseKey(`$H, x, -\infty$`)` aus, anschließend entfernt die Operation `ExtractMin(`H`)` den Knoten. $\rightarrow$ Laufzeit: $\Theta(\log(n))$

Frage: Welche grundsätzliche Idee liegt den Fibonacci-Heaps zugrunde?

Antwort:

- verzichte beim Einfügen + Löschen auf die Vereinigung der Bäume
- hole dies erst bei ExtractMin nach
- vermerke nach jeder Operation das minimale Element

Frage: Welche Struktur und welche Eigenschaften haben Fibonacci-Heaps und was müssen wir bei der Implementierung berücksichtigen?

Antwort:

- Sammlung heap-geordneter Bäume $\rightarrow$ Min-Heap
- Struktur implizit durch erklärte Operationen definiert

$\Rightarrow$ jede mit den bereitgestellten Operationen aufbaubare Struktur ist ein Fibonacci-Heap

Implementierung:

- die Wurzeln der Bäume sind doppelt zyklisch verkettet
- Zeiger auf das kleinste Element der Wurzelliste
- Kinder der Knoten ebenfalls doppelt zyklisch verkettet
- jeder Knoten hat einen *Rang* und ein *Markierungsfeld*
- *Rang:* entspricht der Anzahl der Kinder; es werden nur Bäume gleichen Rangs verschmolzen, damit „breite“ Bäume entstehen (siehe ExtractMin)

- *Markierungsfeld:* es sollen nicht zuviele Knoten eines Teilbaums gelöscht werden, damit die Bäume nicht zu „schmal“ werden (siehe DecreaseKey)

Frage: Eine wichtige Operation bei den Fibonacci-Heaps ist ExtractMin(H), also das Entfernen des minimalen Elements. Wie funktioniert das?

Antwort:

- Entferne den Minimalknoten u aus der Wurzelliste und bilde eine neue Wurzelliste durch Einhängen der Liste der Kinder von u an Stelle von u. $\rightarrow$ Laufzeit: $\Theta(1)$
- Verschmelze nun solange zwei heap-geordnete Bäume, deren Wurzeln denselben Rang haben, zu einem neuen heap-geordneten Baum, bis die Wurzelliste nur Bäume enthält, deren Wurzeln paarweise verschiedenen Rang haben.

Effiziente Implementierung mittels Rang-Array:

- Verschmelzen zweier Bäume B und B' vom Rang i liefert einen Baum vom Rang $i+1$.
- Ist das Element in der Wurzel v von B größer als das Element in der Wurzel v' von B', dann wird v ein Kind von v' und das Markierungsfeld von v wird auf nicht markiert gesetzt.
- Aktualisiere auch den Minimalzeiger.

Laufzeit:

- $\mathcal{O}(n)$ im worst-case!
  - es sind höchstens n Elemente in der Wurzelliste
  - nach jedem Verschmelzen verschwindet ein Baum

  $\rightarrow$ nach n Verschmelzungen gibt es keine Bäume mehr
- $\mathcal{O}(\log(n))$ amortisiert

Frage: Die andere wichtige Funktion bei Fibonacci-Heaps ist DecreaseKey(H, x, k), also das Verkleinern eines gespeicherten Werts. Wie funktioniert das?

Antwort:

- trenne x von seinem Elter
- verkleinere den Wert auf k
- hänge den heap-geordneten Baum in die Wurzelliste
- aktualisiere den Minimalzeiger

$\rightarrow$ Laufzeit: $\Theta(1)$

Es soll verhindert werden, dass mehr als zwei Kinder von seinem Elter abgetrennt werden:

- Beim Abtrennen eines Knotens p von seinem Elter v wird v markiert.
- War v bereits markiert, wird auch v von seinem Elter v' abgetrennt und v' markiert, usw.
- Alle abgetrennten Teilbäume werden in die Wurzelliste aufgenommen, die Markierungen der jeweiligen Wurzeln werden gelöscht.

Laufzeit:

- $\mathcal{O}(\log(n))$ im worst-case
  - die Höhe der Bäume ist logarithmisch beschränkt
  - wird erst bei der amortisierten Laufzeitanalyse gezeigt
- $\mathcal{O}(1)$ amortisiert

Frage: Wie funktionieren die anderen, einfachen Operationen bei Fibonacci-Heaps? Mit Laufzeiten.

Antwort:

- UNION(H_1, H_2): Hänge die Wurzellisten von H_1 und H_2 aneinander und setze den neuen Minimalzeiger auf das Minimum der bisherigen Minimalelemente von H_1 und H_2. $\rightarrow$ Laufzeit: $\Theta(1)$
- INSERT(H, x): Erzeuge Fibonacci-Heap H', der nur x enthält. Der Rang von x ist 0 und das Element ist nicht markiert. Vereinige H und H' mittels UNION. $\rightarrow$ Laufzeit: $\Theta(1)$
- DELETE(H, x): Setze x auf einen sehr kleinen Schlüssel und führe dann EXTRACT-MIN(H) aus. $\rightarrow$ Laufzeit: $\mathcal{O}(n)$ im worst-case, $\mathcal{O}(\log(n))$ amortisiert

8 Suchbäume

Frage: Warum macht es Sinn, Bäume zu balancieren?

Antwort: Ohne Balancierung würde durch das Einfügen von aufsteigend sortierten Werten ein zu einer Liste degenerierter Baum entstehen. Beim Einfügen eines Wertes muss also immer bis ans Ende der Liste gelaufen werden. Als Laufzeit zum Einfügen von n Elementen in einen solchen Baum erhalten wir daher:

$$T(n) = \sum_{i=1}^{n} i \ \in \ \Theta(n^2)$$

Das Balancieren der Bäume führt zu einer besseren worst-case Laufzeit.

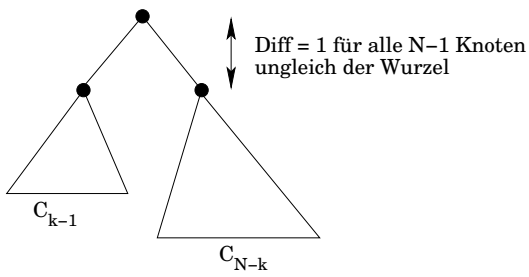
Frage: Wie ist die Laufzeit beim Einfügen zufälliger Werte in einen nicht-balancierten Suchbaum?

Antwort: Beim erfolgreichen Suchen eines Wertes werden gerade soviele Vergleiche benötigt, wie der gesuchte Knoten von der Wurzel entfernt ist. Die interne Pfadlänge (internal path length) eines Baums ist:

$$\sum_{v} \text{Distanz von Knoten } v \text{ zur Wurzel}$$

Wenn wir die interne Pfadlänge durch die Anzahl der Knoten N im Baum dividieren, so erhalten wir die durchschnittliche Anzahl der benötigten Vergleiche. Sei C_N die durchschnittliche interne Pfadlänge eines binären Suchbaums mit N Knoten. Dann gilt:

$$\begin{aligned} C_1 &= 1 \\ C_N &= N - 1 + \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k}) \end{aligned}$$

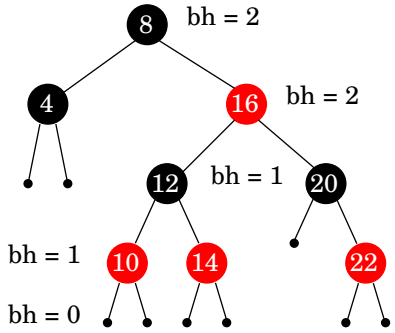


Analog zu der Rekursionsformel von Quicksort gilt $C_N \in \Theta(N \log N)$. Dividieren wir durch N so erhalten wir die durchschnittliche Anzahl Vergleiche bei einer erfolgreichen Suche: $\Theta(\log N)$

Frage: Welche Idee liegt den Rot-Schwarz-Bäumen zu Grunde?

Antwort: Balancierte Binärbäume, die im Gegensatz zu AVL-Bäumen nur ein zusätzliches Bit pro Knoten als Zusatzinformation benötigen:

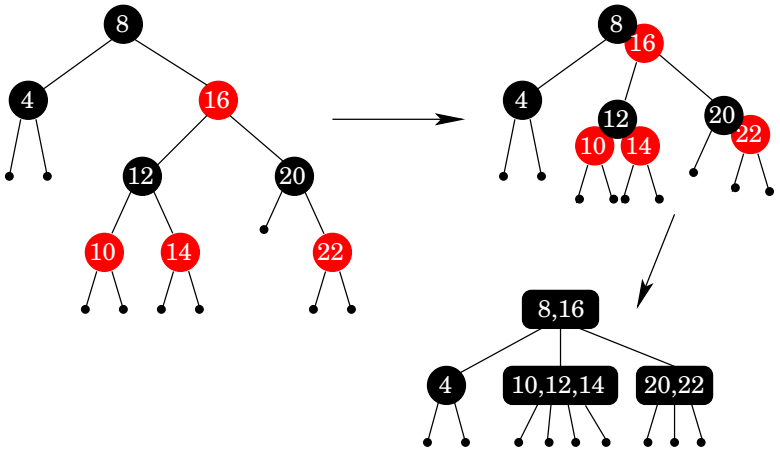
- Jeder Knoten des Baums ist entweder rot oder schwarz.
- Die Wurzel und die Blätter (NIL's) sind schwarz.
- Wenn ein Knoten rot ist, dann ist sein Vorgänger schwarz.
- Jeder einfache, abwärtsgerichtete Weg von einem Knoten x zu einem Blatt hat die gleiche Anzahl schwarzer Knoten: $bh(x)$.



Werden durch das Einfügen oder das Löschen eines Knotens die Bedingungen verletzt, dann werden Knoten umgefärbt und wie bei AVL-Bäumen Rotationen durchgeführt.

Frage: Erklären Sie, warum die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten in $\mathcal{O}(\log(n))$ ist.

Antwort: Verschmelze die roten Knoten mit deren schwarzen Eltern.



Der so entstandene, balancierte Baum der Höhe h hat mindestens 2^h viele Knoten, ist also logarithmisch in der Höhe beschränkt. Die Höhe des originalen Baums ist höchstens doppelt so groß, da niemals zwei rote Knoten aufeinander folgen.

Frage: Wodurch unterscheiden sich B/B*-Bäume von AVL- oder Rot-Schwarz-Bäumen?

Antwort: Es sind keine Binärbäume! Ein B-Baum der Ordnung m hat folgende Eigenschaften:

- Alle Blätter befinden sich in gleicher Tiefe.
- Alle inneren Knoten außer der Wurzel haben mindestens $\lceil m/2 \rceil$ Kinder. In einem nicht leeren Baum hat die Wurzel mindestens 2 Kinder.
- Jeder innere Knoten hat höchstens m Kinder. Ein Knoten mit k Kindern speichert $k-1$ Schlüsselwerte.
- Alle Schlüsselwerte eines Knotens sind aufsteigend sortiert.

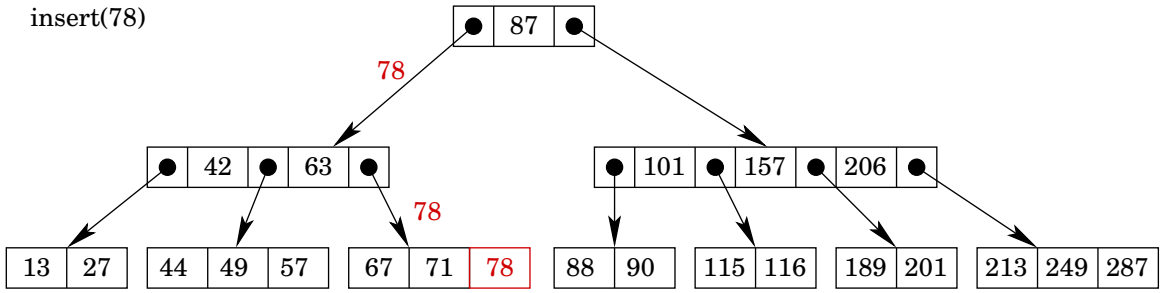
Seien $k_1, \ldots, k_s$ die Schlüssel eines inneren Knotens. Dann gibt es die Zeiger $z_0, z_1, \ldots, z_s$ auf die Kinder und es gilt:

- z_0 zeigt auf einen Teilbaum mit Werten kleiner als k_1.
- z_i für $i = 1, \ldots, s-1$ zeigt auf einen Teilbaum mit Werten größer als k_i und kleiner als k_{i+1}.
- z_s zeigt auf einen Teilbaum mit Werten größer als k_s.

Einfügen eines Wertes: Bestimme zunächst mittels einer Suche das Blatt, in dem der Wert abgelegt werden muss. Wir unterscheiden zwei Fälle:

- Fall 1: Das Blatt hat noch nicht die maximale Anzahl $m-1$ von Schlüsseln gespeichert.

  Dann fügen wir den Wert entsprechend der Sortierung ein.

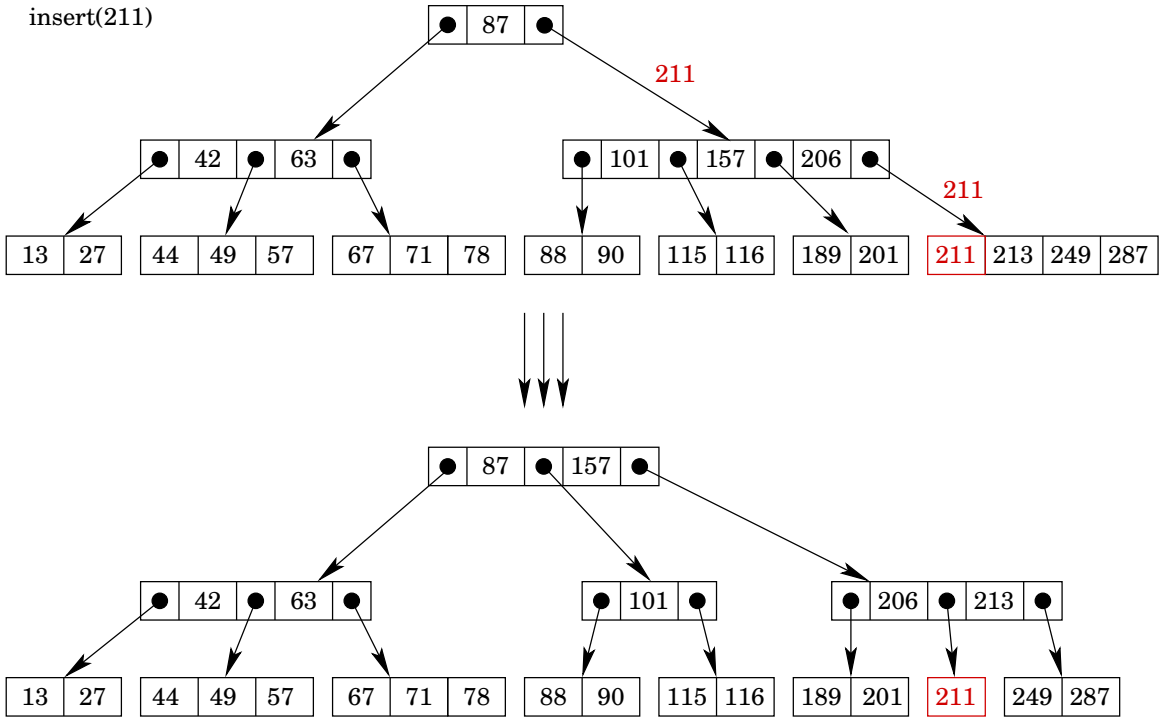


- Fall 2: Das Blatt hat bereits die maximale Anzahl $m-1$ von Schlüsseln gespeichert.

  In diesem Fall ordnen wir den Wert wie im Fall 1 entsprechend seiner Größe ein und teilen anschließend den zu groß gewordenen Knoten in der Mitte auf. Das mittlere Element wird in den Vorgänger-Knoten eingefügt.

  Dieses Teilen wird solange längs des Suchpfades bis zur Wurzel fortgesetzt, bis ein Knoten erreicht ist, der noch nicht die maximale Anzahl von Schlüsseln speichert, oder bis die Wurzel erreicht wird.

  Muss die Wurzel geteilt werden, so schafft man eine neue Wurzel, die den mittleren Schlüssel als einzigen Schlüssel speichert.



Frage: Was ist das Besondere an Splay-Bäumen?

Antwort: Strukturanpassung an unterschiedliche Zugriffshäufigkeiten:

- Oft angefragte Schlüssel werden in Richtung Wurzel bewegt.
- Selten angefragte Schlüssel wandern zu den Blättern hinab.
- Die Zugriffshäufigkeiten sind vorher nicht bekannt.

Sei `T` ein Suchbaum und `x` ein Schlüssel. Dann ist `splay(T,x)` der Suchbaum, den man wie folgt erhält:

- Schritt 1: Suche nach `x` in `T`. Sei `p` der Knoten, bei dem die erfolgreiche Suche endet, falls `x` in `T` vorkommt. Ansonsten sei `p` der Vorgänger des Blattes, an dem die Suche nach `x` endet, falls `x` nicht in `T` vorkommt.
- Schritt 2: Wiederhole die Operationen `zig` (einzelne Rotation), `zig-zig` (zwei Rotationen in die gleiche Richtung) und `zig-zag` (zwei Rotationen in entgegengesetzte Richtungen) beginnend bei `p` solange, bis sie nicht mehr ausführbar sind, weil `p` Wurzel geworden ist.

Frage: Was bewirkt die Splay-Operation?

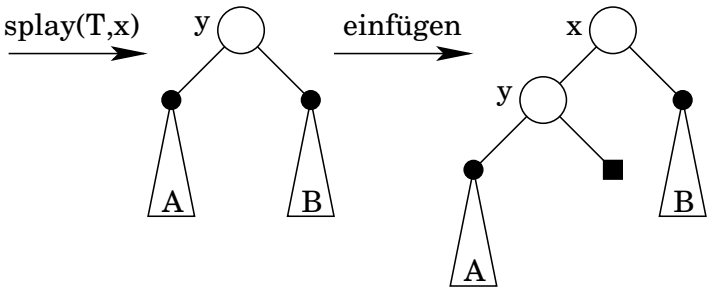
Antwort:

- Kommt `x` in `T` vor, so erzeugt `splay(T,x)` einen Suchbaum, der den Schlüssel `x` in der Wurzel speichert.
- Kommt `x` nicht in `T` vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel `x` unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

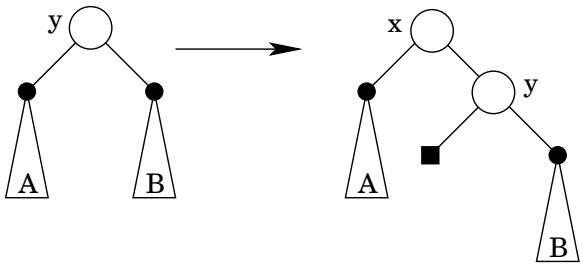
Frage: Wie funktioniert ein Insert bei Splay-Bäumen?

Antwort: Um `x` in `T` einzufügen, rufe `splay(T,x)` auf. Ist `x` nicht in der Wurzel, so füge wie folgt eine neue Wurzel mit `x` ein. Beachte obige zweite Aussage: Kommt `x` nicht in `T` vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel `x` unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

- Falls der Schlüssel der Wurzel von `T` kleiner als `x` ist:



- Falls der Schlüssel der Wurzel von `T` größer als `x` ist:

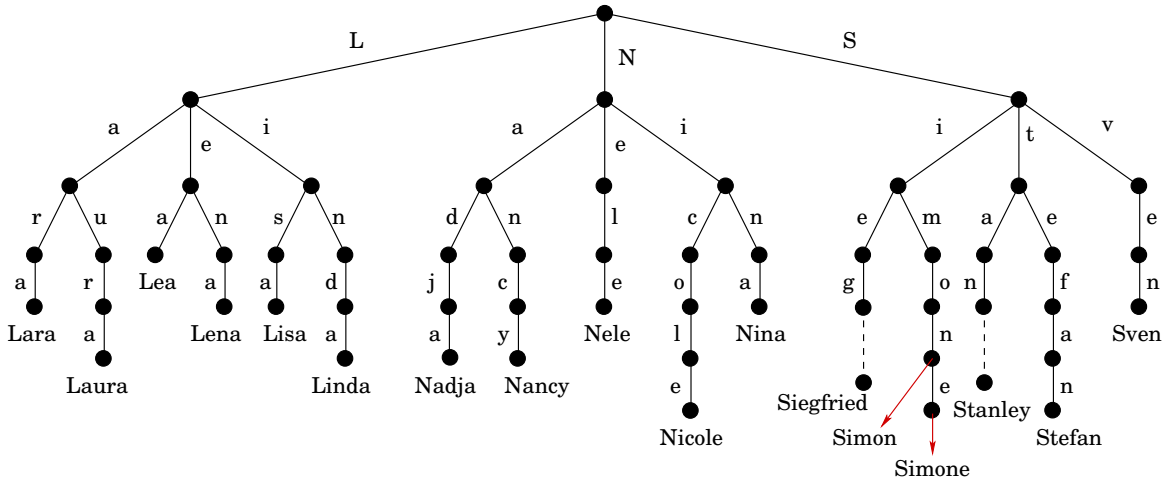


Frage: Was sind Tries? Welche Struktur haben sie und wofür werden sie eingesetzt?

Antwort: Tries sind Digitale Suchbäume:

- Ein Schlüssel, der aus einzelnen Zeichen eines Alphabets besteht, wird in Teile zerlegt.

- Der Baum wird anhand der Schlüsselteile aufgebaut: Bits, Ziffern oder Zeichen eines Alphabets oder Zusammenfassung der Grundelemente wie Silben der Länge k.
- Eine Suche im Baum nutzt nicht die totale Ordnung der Schlüssel, sondern erfolgt durch Vergleich von Schlüsselteilen.
- Jede unterschiedliche Folge von Teilschlüsseln ergibt einen eigenen Suchweg im Baum.
- Alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg. Digitale Suchbäume werden daher auch Präfix-Bäume genannt.

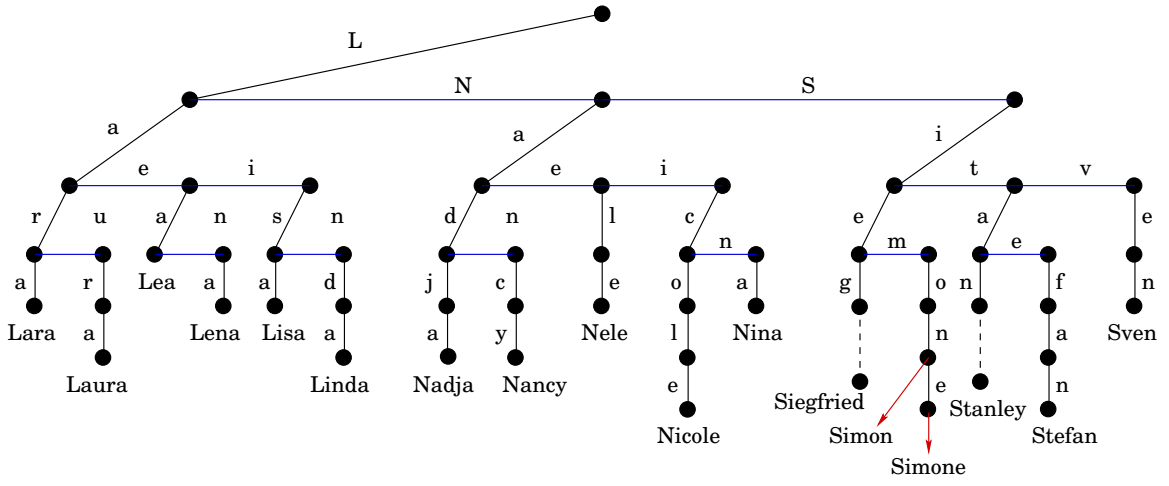


Beobachtungen:

- Die Höhe eines Tries wird durch den längsten abgespeicherten Schlüssel bestimmt.
- Die Gestalt des Baumes hängt von der Verteilung der Schlüssel ab, nicht von der Reihenfolge ihrer Abspeicherung.
- Knoten, die nur `NULL`-Zeiger besitzen, werden nicht angelegt.
- Schlechte Speicherplatzausnutzung aufgrund dünn besetzter Knoten und vieler Einweg-Verzweigungen in der Nähe der Blätter.

Frage: Ein Problem bei der Realisierung von Tries sind die hohen Knotengrade und die schlechte Speicherplatzausnutzung bei dünn besetzten Bäumen. Gibt es alternative Implementierungen?

Antwort: Realisierung als binärer Baum: Speichere in jedem Knoten nur Verweise auf zwei Knoten: Einen zum ersten Kind und einen zum Geschwister.



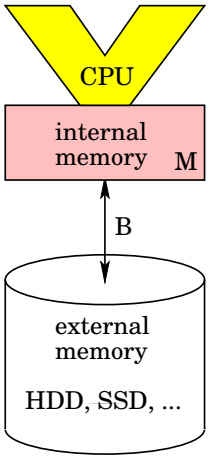
Problem: Zugriff auf Kind mit bestimmten Namen nicht mehr in konstanter Zeit möglich. Daher wird zusätzlich Hashing angewendet:

- Speichere nur Referenzen auf Eltern-Knoten.
- Nutze Hash-Funktion, um zu einem Kind mit einem bestimmten Namen zu gelangen.
- Die Größe der Hash-Tabelle richtet sich nach der Größe des Wörterbuchs.

9 Algorithmen für moderne Hardware

Frage: Um die komplexe Speicherhierarchie in einem vereinfachten Modell zu beschreiben, haben wir das *externe Speichermodell* kennengelernt. Beschreiben Sie dieses Modell.

Antwort: Starke Abstraktion der Speicherhierarchie auf nur 2 Ebenen.



Parameter des Modells:

- M: maximale Anzahl Datenelemente in dem internen Speicher
- B: Anzahl Datenelemente, die mit einer I/O-Operation zwischen dem internen und dem externen Speicher ausgetauscht werden kann

Annahmen für die Performanceanalyse:

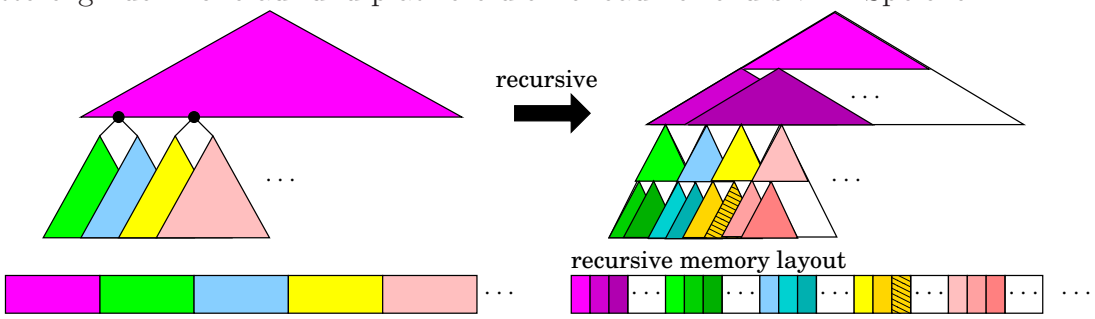
- Bandbreite ist unendlich und damit nicht der Flaschenhals der Auslagerung
- Datenzugriff auf internen Speicher kostet eine Zeiteinheit
- arithmetische Operationen kosten eine Zeiteinheit

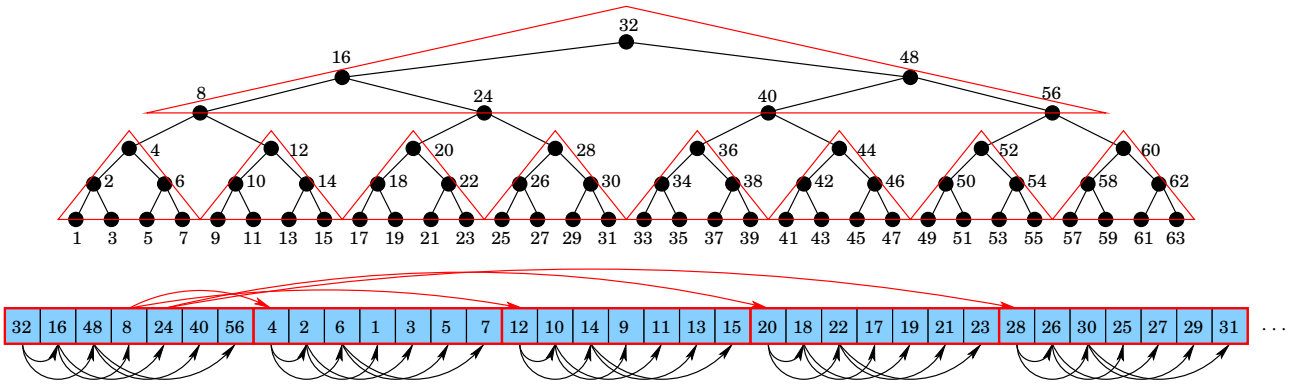
$\rightarrow$ Anzahl der I/O-Operationen ist aussagekräftig

Externer Speicher kann auch RAM bezeichnen, dann ist der interne Speicher der Cache. Das Modell kann also auch genutzt werden, um Cache-Effekte zu untersuchen.

Frage: Um bei der binären Suche die Lokalität der Daten nutzen zu können, haben wir das Layout nach *van Emde Boas* kennengelernt. Beschreiben Sie dieses Layout der Daten.

Antwort: Erstelle zunächst den vollständig balancierten Suchbaum, teile dann den Baum in der Mitte bzgl. der Höhe auf und platziere die Teilbäume rekursiv im Speicher.



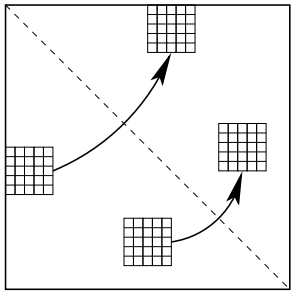


Frage: Wie haben wir beim Transponieren von (quadratischen) Matrizen Lokalität der Daten genutzt?

Antwort: Zur Cache-Größe C definieren wir eine Blockgröße B, sodass $2B^2 \leq C$ gilt und daher beide Teilmatrizen in den Cache passen.

- Durchlaufe die Teilmatrizen der Größe $B \times B$ von A und transponiere diese Teilmatrizen.
- 1 Block von A lesen $\rightarrow B$ viele I/Os
- 1 Block nach A^T schreiben $\rightarrow B$ viele I/Os
- insgesamt gibt es $n/B \cdot n/B$ viele Teilmatrizen

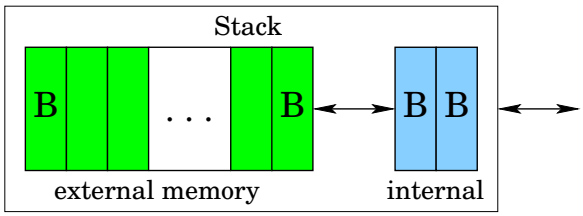
$\rightarrow$ $n^2/B^2 \cdot 2B = 2n^2/B \in \mathcal{O}(n^2/B)$ viele I/Os



```
for (int i = 0; i < n; i += blksize)
    for (int j = 0; j < n; j += blksize)
        // transpose the block beginning at [i,j]
        for (int k = i; k < i + blksize; ++k)
            for (int l = j; l < j + blksize; ++l)
                AT[l][k] = A[k][l];
```

Frage: Wie wird ein Stack als externe Datenstruktur realisiert?

Antwort: Nutze kombinierten Input/Output-Buffer der Größe $2 \cdot B$:

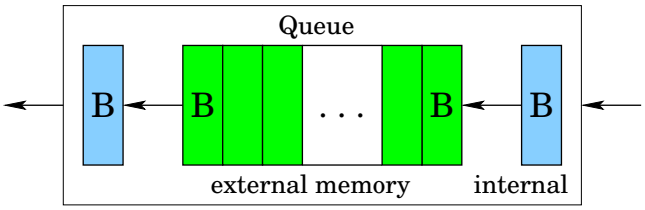


- push: Füge neues Element in den Buffer ein. Falls der Buffer voll ist, dann schreibe den unteren Block auf die Festplatte. $\rightarrow 1/B$ viele I/Os amortisiert
- top/pop: Lese/entferne Element aus dem Buffer. Falls der Buffer leer ist, lese den zuletzt geschriebenen Block von der Festplatte in den Buffer ein. $\rightarrow 1/B$ viele I/Os amortisiert

---

Frage: Wie wird eine Queue als externe Datenstruktur realisiert?

Antwort: Nutze getrennte Input- und Output-Buffer jeweils der Größe B:

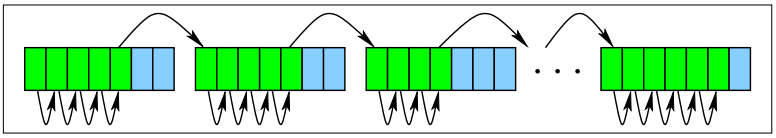


- put: Füge neues Element am Ende des Input-Buffers ein. Falls der Input-Buffer voll ist, schreibe den Buffer auf die Festplatte. $\rightarrow {}^1/_B$ viele I/Os amortisiert
- head/get: Lese/entferne erstes Element aus dem Output-Buffer. Falls der Output-Buffer leer ist, dann lese Block von der Festplatte oder, falls leer, aus dem Input-Buffer. $\rightarrow {}^1/_B$ viele I/Os amortisiert

---

Frage: Wie werden lineare Listen als externe Datenstruktur realisiert?

Antwort: Blöcke müssen nicht komplett gefüllt sein, aber stelle sicher, dass *jedes Paar konsekutiver Blöcke* mindestens ${}^2/_3B$ viele Elemente enthält. $\rightarrow$ maximal ${}^{3n}/_B \in \mathcal{O}({}^n/_B)$ viele Blöcke werden zur Speicherung der n Werte benötigt



insert:

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}({}^n/_B)$ viele I/Os
- das eigentliche Einfügen kostet meist nur 1 I/O, außer wenn Block voll ist, dann:
  - falls benachbarter Block noch Platz hat: tausche ein Element aus $\rightarrow \mathcal{O}(1)$ viele I/Os
  - falls beide Nachbarblöcke voll sind: teile den Block in 2 Teile der Größe $\approx B/2$ mit $\mathcal{O}(1)$ vielen I/Os. Danach sind mindestens ${}^B/_6$ Lösch-Operationen nötig, um Invariante zu verletzen.

$\rightarrow \mathcal{O}(1 + {}^n/_B)$ viele I/Os

erase:

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}({}^n/_B)$ viele I/Os
- das eigentliche Löschen kostet meist nur 1 I/O, außer wenn der Block danach mit einem der beiden benachbarten Blöcke weniger als ${}^2/_3B$ Elemente besitzt
- dann: verschmelze die beiden Blöcke $\rightarrow \mathcal{O}(1)$ viele I/Os

$\rightarrow \mathcal{O}(1 + {}^n/_B)$ viele I/Os

Frage: Zu obiger linearer Liste: Warum nicht einfach fordern, dass jeder Block mindestens zu einem Drittel gefüllt sein muss? Warum die „seltsame“ Regel mit Paaren von konsekutiven Blöcken?

Antwort: Beim Löschen von Elementen müsste der Block, aus dem das Element gelöscht wurde, mit einem benachbarten Block verschmolzen werden, sobald nicht mehr genug Elemente im Block vorhanden sind.

Das Zusammenfassen geht nicht, falls in den Nachbarblöcken zu viele Elemente enthalten sind. Dann müsste wieder gesplittet werden usw. Daher muss der Füllgrad nur für benachbarte Paare von Blöcken gelten.

Frage: Wir hatten in der Vorlesung gesehen, dass Hashing mit Verkettung der Überläufer eine gute erwartete Performance im externen Speicher aufweist. Skizzieren Sie die Analyse.

Antwort: Um die mittlere Laufzeit bei erfolgreicher Suche abschätzen zu können, gehen wir davon aus, dass die N Einträge durch die Hash-Funktion h gleichmäßig auf die M Listen verteilt werden. Dann ist N/M die durchschnittliche Anzahl der Einträge in $h(k)$.

- Beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge $^{j-1}/_{M}$.
- Eine spätere Suche nach dem j-ten Schlüssel betrachtet im Schnitt $1 + {}^{j-1}/_{M}$ Einträge, wenn stets am Listenende eingefügt wird und keine Elemente gelöscht wurden.

$$S(\alpha) = \frac{1}{N}\sum_{j=1}^{N}\left(1 + \frac{j-1}{M}\right) = 1 + \frac{1}{NM}\cdot\frac{(N-1)N}{2} = 1 + \frac{N-1}{2M} \approx 1 + \frac{N}{2M} = 1 + \frac{\alpha}{2}$$

In der Regel gilt $N \in \mathcal{O}(M)$: Die Größe der Hashtabelle ist ungefähr so groß wie die Anzahl der Datensätze. $\rightarrow \alpha = {}^{N}/_{M} = {}^{\mathcal{O}(M)}/_{M} \in \mathcal{O}(1)$

Zum Vergleich: Binäre Suche hat Laufzeit $\mathcal{O}(\log(N))$ plus Aufwand $\mathcal{O}(N \cdot \log(N))$ zum Sortieren der Datensätze.

10 Amortisierte Laufzeitanalyse

Frage: Als einführendes Beispiel haben wir einen Stack betrachtet, der mittels Arrays implementiert war. Das Array wurde bei Bedarf vergrößert. Die Laufzeit von `push` im worst-case war $\mathcal{O}(n)$: Es musste Speicherbereich für $2n$ Elemente allokiert werden, dann mussten n Elemente in den größeren Speicherbereich verschoben werden, und anschließend wurde das neue Element in den Stack eingetragen. Wie konnten wir diese Abschätzung „verbessern“?

Antwort: N Ausführungen von `push` dauern bei der „normalen“ Abschätzung:

$$\sum_{i=1}^{N}\mathcal{O}(i) \;=\; \mathcal{O}(N^2)$$

Aber: Ein `push` ist nur selten teuer, deshalb genauere Abschätzung für N Ausführungen von `push`:

$$\sum_{i=1}^{N} 1 + \sum_{i=1}^{\log_2(N)} 2^i \;\leq\; N + 2^{\log_2(N)+1} \;=\; 3 \cdot N \;\in\; \mathcal{O}(N)$$

Amortisierte (durchschnittliche) Laufzeit von `push`:

$$T(N) \;=\; \frac{\mathcal{O}(N)}{N} \;=\; \mathcal{O}(1)$$

Frage: Schätzen Sie am Beispiel der Account-Methode die amortisierte Laufzeit der `push`-Operation ab.

Antwort: Die Account-Methode wird auch Bankkonto-Paradigma genannt. Man betrachtet eine beliebige Folge von n Operationen, wobei jede Operation Geld kostet. Dabei werden die tatsächlichen Kosten der i-ten Operation mit t_i und die amortisierten Kosten der i-ten Operation mit a_i bezeichnet. Falls $t_i < a_i$ ist, schreibe den Überschuss auf dem Konto gut, falls

$t_i > a_i$ ist, zahle die Differenz vom Konto aus. Wichtig: Der Kontostand darf niemals negativ sein!

Pro `push` bezahle 3 elementare Einfüge-Operationen: Einfügen des Elements, verschieben des Elements beim Vergrößern sowie verschieben eines anderen Elements beim Vergrößern.

Der Kontostand wird dabei niemals negativ! Nur bei der Vergrößerung des Speicherbereichs könnte der Kontostand negativ werden, aber:

- zwischen den beiden Vergrößerungen auf 2^k und 2^{k+1} Elemente wurden genau 2^k Elemente eingefügt
- auf dem Konto ist also mindestens ein Guthaben von $2^k \cdot (3-1) = 2^{k+1}$

$\rightarrow$ entspricht genau der Anzahl Elemente, die verschoben werden müssen

---

Frage: Wir haben auch die Potenzial-Methode kennengelernt. Schätzen Sie damit die amortisierten Kosten der `push`-Operation ab.

Antwort: Anstelle von Kontoständen weise der Datenstruktur eine potenzielle Energie zu. Dabei ist Φ_i das Potenzial nach Ausführung der i-ten Operation.

Betrachte eine Folge von n Operationen. Ordne jedem Bearbeitungszustand ein nicht-negatives Potenzial zu, und ordne jeder Operation amortisierte Kosten zu. Die amortisierten Kosten a_i der i-ten Operation sind die tatsächlichen Kosten t_i plus der Differenz der Potenziale

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Wenn wir die Kosten aufsummieren, erhalten wir folgende Teleskop-Summe:

$$\sum_{i=1}^{n} a_i = (t_1 + \Phi_1 - \Phi_0) + (t_2 + \Phi_2 - \Phi_1) + \ldots + (t_n + \Phi_n - \Phi_{n-1}) = \Phi_n - \Phi_0 + \sum_{i=1}^{n} t_i$$

In der Praxis wählen wir eine Potenzialfunktion, bei der $\Phi_0 = 0$ ist, und die niemals negativ wird. Dann sind die amortisierten Kosten eine obere Schranke für die tatsächlichen Kosten. Für den Stack S definiere die Potenzial-Funktion Φ als:

$$\Phi(S) := 2 \cdot num(S) - size(S)$$

Dabei ist num die Anzahl der gespeicherten Elemente und $size$ die Größe des Arrays. Wir unterscheiden zwei Fälle beim Einfügen des i-ten Elements: Wird das Array nicht vergrößert, dann gilt $size_i = size_{i-1}$, sonst gilt $size_i/2 = size_{i-1} = num_i - 1$. Setzt man dies jeweils in die Formel $a_i = t_i + \Phi_i - \Phi_{i-1}$ ein, so erhält man konstante Kosten für die amortisierte Laufzeit.

Frage: Wie haben wir bei den selbstanordnenden linearen Listen gezeigt, dass die Move-To-Front-Regel höchstens doppelt so schlecht ist, wie jede andere Regel zur Selbstanordnung von Listen.

Antwort: Wir haben die Anzahl der Inversionen vor und nach Ausführung der i-ten Operation gezählt. Für die *Potenzialfunktion* $\Phi_i = inv(L_{A,i}, L_{MF,i})$ gilt:

- Initial ist $\Phi_0 = inv(L_{A,0}, L_{MF,0}) = 0$, da beide Algorithmen mit derselben Liste beginnen.
- Der Wert der Potenzialfunktion ist nie negativ!

Frage: Um die Höhe der Bäume in den Fibonacci-Heaps abschätzen zu können, hatten wir folgendes festgestellt: Ein Knoten x mit $deg(x) = k$ ist die Wurzel eines Baums mit mindestens F_k Knoten. Stellen Sie die Idee des Beweises dar.

Antwort: Wir hatten bei den Operationen auf dem Heap sichergestellt, dass nicht „zu viele“ Kinder von einem Knoten abgetrennt werden. Seien $y_1, \ldots, y_k$ die Kinder von x in der zeitlichen Reihenfolge, in der sie an x angehängt wurden. Dann gilt:

$$deg(y_1) \geq 0 \quad \text{und} \quad deg(y_i) \geq i - 2 \text{ für } i = 2, 3, \ldots, k$$

Induktionsanfang: $deg(y_1) \geq 0$ ist klar.

Induktionsschluss für $i \geq 2$:

- Als y_i an x angehängt wurde, waren $y_1, \ldots, y_{i-1}$ Kinder von x, also gilt $deg(x) \geq i - 1$.
- Da y_i an x angehängt wird, gilt: $deg(x) = deg(y_i)$, also gilt $deg(y_i) \geq i - 1$.
- Seither hat Knoten y_i höchstens ein Kind verloren, sonst hätten wir y_i von x abgetrennt, also gilt: $deg(y_i) \geq i - 2$

Aus der Mathematik wissen wir, dass $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ gilt. Mit Hilfe dieser beiden Tatsachen können wir mittels vollständiger Induktion zeigen, dass $size(x) \geq F_{k+2}$ gilt.

Frage: Welche Kenngrößen gehen in die Potenzialfunktion zur Laufzeitabschätzung der Operationen auf Fibonacci-Heaps ein?

Antwort: Die Potenzial-Funktion zu einem Fibonacci-Heap H ist definiert als

$$\Phi(H) = b(H) + 2 \cdot m(H)$$

Dabei bezeichnet $b(H)$ die Anzahl der Bäume in der Wurzelliste und $m(H)$ die Anzahl der markierten Knoten.

Frage: Bei der Union-Find-Datenstruktur mit Vereinigung nach Höhe/Größe und Pfadkomprimierung hatten wir festgestellt, dass die Laufzeiten der `union-` und der `find`-Operation $\mathcal{O}(\log^*(n))$ beträgt. Was bedeutet $\log^*(n)$?

Antwort: $\log^*(n)$ gibt an, wie oft man die log-Taste des Taschenrechners nach Eingabe von n drücken muss, bis ein Wert kleiner als 2 angezeigt wird. Oder mathematisch:

$$\log^{(i)}(n) := \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)}(n)) & \text{für } i \geq 1 \end{cases}$$

Damit definieren wir:

$$\log^\star(n) := \min\{i \geq 0 \mid \log^{(i)}(n) < 2\}$$

Frage: Skizzieren Sie die Idee der Laufzeit-Abschätzung für die `union-` und die `find`-Operation bei der Union-Find-Datenstruktur mit Vereinigung nach Höhe/Größe und Pfadkomprimierung.

Antwort: Zur Laufzeitabschätzung ordnen wir den Knoten Blöcke zu. Ein Knoten x wird dem Block b zugeordnet, wenn gilt:

$$tower(b-1) < rank(x) \leq tower(b)$$

Dabei definieren wir *tower* als Umkehrfunktion der obigen $\log^\star$-Funktion:

$$tower(b) := \left.2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}\right\} b$$

Betrachte eine `find`-Operation: Es wird ein Weg $x_1, x_2, x_3, \ldots, x_l$ durchlaufen, wobei x_1 der Parameter der `find`-Operation ist, und x_l der Repräsentant des Baums bzw. der Menge.

- Die Kosten von `find`(x_1) entsprechen der Länge l des Weges.
- Jeder Knoten auf dem Weg zahlt jeweils 1$ in eines von mehreren Konten.
  - leader account: darin zahlen Repräsentanten ein
  - child account: darin zahlen Kinder der Repräsentanten ein
  - block accounts: darin zahlen Knoten x_k ein, die in einem anderen Block sind als deren Vorgänger x_{k+1}
  - path account: darin zahlen alle anderen Knoten ein
- Der Gesamtbetrag aus allen Konten gibt dann die Summe der Laufzeiten aller Operationen an.

Jede `find`-Operation zahlt

- 1$ in das Repräsentanten-Konto (leader account),
- höchstens 1$ in das Kinder-Konto (child account), und
- höchstens 1$ in jedes Block-Konto (block account), wovon es nur $\log^\star(n)$ viele gibt.

Bei m Operationen über n Elementen wird insgesamt höchstens ein Wert von $2m + m \cdot \log^\star(n)$ in diese Konten eingezahlt. Um abzuschätzen, wieviel in das Pfad-Konto eingezahlt wird, muss etwas gerechnet werden. Wir erhalten $n \cdot \log^\star(n)$ als Wert, und daher $\mathcal{O}(\log^\star(n))$ als Wert für jede einzelne Operation.