Aufgaben

Kim Thuong Ngo

July 13, 2018

Contents

1 Blatt 0

1.1 Logarithmus

A) Zeigen Sie: $b^{log_b(a)} = a$
$log_b(a) = x \Leftrightarrow b^x = a$
Sei $x = log_b(a)$ für angemessenes $x \epsilon \mathbb{R} \Rightarrow b^x = a \Rightarrow a = b^x = b^{log_b(a)}$

B) Zeigen Sie: $log_b(x * y) = log_b(x) + log_b(y)$
Sei $x_1 = log_b(x), x_2 = log_b(y) \Rightarrow b^{x_1} = xb^{x_2} = y \Rightarrow x * y = b^{x_1} * b^{x_2} = b^{x_1+x_2} \Rightarrow log_b(x * y) = x_1 + x^2 = log_b(x) + log_b(y)$

C) Berechnen Sie $2^{log_4(n)}$
$2^{log_4(n)} = 2^{log_2(n^{\frac{1}{2}})} = n^{\frac{1}{2}} = \sqrt{n}$

1.2 Summenformel

A) Kleiner Gauss $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$

B) $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$

C) $\sum_{i=1}^{n}(i2^i) = 2 + 2^{n+1}(n-1)$

1.3 Ereignisraum und Ereignisse

Lösen Sie folgende Aufgaben unter der Annahme, dass Ereignisse gleichverteilt sind.
Münze: Kopf (K), Zahl (Z)

A) Bestimmen Sie den Ereignisraum für: "Eine Münze wird drei Mal hintereinander geworfen." Betrachten Sie das Ereignis: "Es wird mindestens zwei Mal Kopf geworfen. " Wie sieht dieses Ereignis als Menge geschrieben aus? Wie ist die Wahrscheinlichkeit für dieses Ereignis?
Münze wird dreimal geworfen: $\Omega = KKK, KKZ, KZK, ZKK, ZZK, ZKZ, KZZ, ZZZ$ mind. zweimal Kopf: $S = KKK, KKZ, KZK, ZKK$ $P[S] = \frac{|S|}{|\Omega|} = \frac{4}{8} = \frac{1}{2}$

B) Zeigen Sie: Aus $A \cap B = \varnothing$ folgt $P[A \cup B] = P[A] + P[B]$. Was gilt, wenn $A \cap B \neq \varnothing$?
$A \cap B = \varnothing \Rightarrow P[A \cup B] = P[A] + P[B]$ $|A \cup B| = |A| + |B| - |A \cap B|$ $A \cap B = \varnothing \Rightarrow |A \cap B| = 0$
$P[A \cup B] = \frac{|A \cup B|}{|\Omega|} = \frac{|A| + |B|}{|\Omega|} = \frac{|A|}{|\Omega|} + \frac{|B|}{|\Omega|} = P[A] + P[B]$

C) Zeigen Sie für das Gegenereignis $A^C = \Omega \backslash A$ eines Ergebnisses A: $P[A^C] = 1 - P[A]$.
$A^C = \Omega \backslash A : P[A^C] = 1 - P[A]$ $\Omega = A \cup A^C, A \cap A^C = \varnothing \Rightarrow P[\Omega] = P[A \cup A^C] = P[A] + P[A^C]$ da $P[\Omega] = 1 \Rightarrow P[A^C] = 1 - P[A]$

1.4 Zufallsvariable

Eine Zufallsvariable X ist eine Funktion $X : \Omega \rightarrow M$, wobei Ω ein Ereignisraum ist und M eine beliebige Menge.

Sei $\Omega = 1, ..., 10^2$. Betrachten Sie die Zufallsvariable

$$X : \Omega \rightarrow \mathbb{N}, X(x, y) = x + y.$$

Wir definieren das Ereignis $[X \leq a] = (x, y) \epsilon \Omega | X(x, y) \leq a$.

A) Geben Sie die Menge $[X \leq 5]$ konkret an und beschreiben Sie das Ereignis in Worten.
$[X \leq 5] = (1,1), (1,2), (1,2), (1,4), (2,1), (2,2), (2,3), (3,1), (3,2), (4,1)$

B) Berechnen Sie $P[X \leq 5]$ unter Annahme der Gleichverteilung der Ereignisse.
$P(X \leq 5)$ bei Gleichverteilung: $|[X \leq 5]| = 10$ $|\Omega| = 10 * 10 = 100 \Rightarrow P[X \leq 5] = \frac{|[X \leq 5]|}{|\Omega|} = \frac{10}{100} = \frac{1}{10}$

1.5 Erwartungswert und Varianz

Sei Ω ein Ereignisraum. Wir definieren den Erwartungswert einer Zufallsvariable $X : \Omega \rightarrow M$ als

$$E[X] = \sum_{x \epsilon M} x * P[X = x].$$

Intuitiv beschreibt der Erwartungswert einer Zufallsvariable das Ereignis, welches im Mittel am häufigsten auftritt. Der Erwartungswert ist linear, d.h. es gilt

$$E[a + b * X] = a + b * E[X]$$

A) Berechnen Sie den Erwartungswert einer Zufallsvariable, die nur Werte 0 und 1 haben kann.
$E(X) = \sum_{x \epsilon M} x * P[X = x] = 0 * P[X = 0] + 1 * P[X = 1] = P[X = 1]$

B) Berechnen Sie den Erwartungswert eines fairen Würfels.
Erwartungswert fairer Würfel: $W = 1, ..., 6$ $E(X) = \sum_{x \epsilon M} x * \frac{1}{6} = \frac{1}{6} * \sum_{x \epsilon M} x = \frac{1}{6} * 21 = 3,5$

C) Verwenden Sie die Linearität des Erwartungswertes, um den Erwartungswert der Summe von zwei unabhängigen Würfelwürfen zu berechnen.
X: Ergebnis 1. Wurf Y: Ergebnis 2.Wurf $E[X + Y] = E[X] + E[Y] = 3,5 + 3,5 = 7$

D) Die Varianz einer Zufallsvariable X gibt das Mittel der quadratischen Abweichung von X zu ihrem Erwartungswert an. Formal:

$$var(X) = E[(X - E(x))^2].$$

Zeigen Sie mit Hilfe der Linearität des Erwartungswerts, dass folgende Gleichung gilt:

$$var(X) = E[X^2] - E[X]^2.$$

$var(x) = E[(X - E(X))^2] = E[X^2] - E[X]^2 = E[X^2 - 2x * E(X) + E(X)^2] = E[X^2] - E[2XE(X)] + E[E(X)^2] = E[X^2] - 2E(X)E(X) + E(X)^2 = E(X^2) - 2E(X)^2 + E(X)^2 = E(X^2) - E(X)^2$

2 TUTORIUM 23.04.2018: REKURSION

$T(0) = 0$ $T(n) = 2^{n-1} + T(n-1) = 2^{n-1} + 2^{(n-1)-1} + T((n-1)-1)$ ... $2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^{n-i} + T(n-i)$

Vollständige Induktion

IB: $T(n) := \sum_{k=1}^{i} 2^{n-k} + T(n-i)$ IV: Behauptung gelte für beliebige feste $i \epsilon \mathbb{N}$ IA: $T(n)_1 = \sum_{k=1}^{1} 2^{n-k} + T(n-1) = 2^{n-1} + T(n-1)$ IS: Beh: $T(n)_{i+1} = \sum_{k=1}^{i+1} 2^{n-k} + T(n-(i+1))$ $T(n) := \sum_{k=1}^{i} 2^{n-k} + T(n-i) \Rightarrow \sum_{k=1}^{i} 2^{n-i-1} + T(n-i-1) = \sum_{k=1}^{i} 2^{n-k} + 2^{n-(i+1)} + T(n-(i+1))$ $I.V. = \sum_{k=1}^{i+1} 2^{n-k} + T(n-(i+1)) = T(n)_{i+1} \Rightarrow T(n)_n = \sum_{k=1}^{n} 2^{n-k} + T(n-n) = \sum_{k=1}^{n} 2^{n-k}$

In welcher Laufzeitklasse liegt eine rekursive Funktion?

$T(n) = aT(\frac{n}{b}) + f(n); a, b \geq 1; f(n), T(n) \geq 0; \varepsilon > 0$

1.Fall $f(n) \epsilon \mathcal{O}(n^{log_b(a-\varepsilon)})$ für ein $\varepsilon > 0 \Rightarrow T(n) \epsilon \Theta(n^{log_b(a)})$

2.Fall $f(n) \epsilon \Theta(n^{log_b(a)}) \Rightarrow T(n) \epsilon \Theta(n^{log_b(a)} * log_b(n))$

3.Fall $f(n) \epsilon \Omega(n^{log_b(a+\varepsilon)}) \Rightarrow T(n) \epsilon \Theta(f(n))$

- $\mathcal{O}(n)$ Oberschranke
- $\Omega(n)$ Unterschranke
- $\Theta(n)$ Vereinigung beider Schranken

2.1 BEISPIELE

Mergesort $T(n) = 2T(\frac{n}{2}) + n$ $a = 2, b = 2, f(n) = n$ $n^{log_2 2} = n \Rightarrow 2.Fall : T(n) \epsilon \Theta(n * log_2 n)$

binarySearch $T(n) = T(\frac{n}{2}) + 1$ $a = 1, b = 2, f(n) = 1$ $1^{log_2 1} = 1 \Rightarrow 2.Fall$

3 Blatt 01

3.1 O-Notation

A) Aus $f_1(n), f_2(n) = \mathcal{O}(g(n))$ folgt $f_1(n) + f_2(n) = \mathcal{O}(g(n))$ und $f_1(n) \cdot f_2(n) = \mathcal{O}(g(n)^2)$.

B) Aus $f(n) = \mathcal{O}(g(n))$ und $g(n) = \mathcal{O}(h(n))$ folgt $f(n) = \mathcal{O}(h(n))$.

C) $f(n) = \Theta(g(n))$ genau dann, wenn $g(n) = \Theta(f(n))$.

D) $f(n) = \mathcal{O}(g(n))$ genau dann, wenn $g(n) = \Omega(f(n))$.

3.2 Mastertheorem

Bestimmen Sie die Komplexitätsklasse für folgende Rekursionsgleichung mit Hilfe des Mastertheorems:

A) $T(n) = T(\frac{n}{2}) + 1$

- $a = 1$
- $b = 2$
- $f(n) = 1$

$n^{log_2(1)} = n^0 = 1 \Rightarrow 2.Fall\ f(n) \in \Theta(1) \Rightarrow T(n) \in \Theta(logn)$

B) $T(n) = 2T(\frac{n}{2}) + 1$

- $a = 2$
- $b = 2$
- $f(n) = 1$

$n^{log_2(2-\varepsilon)} = n^{log_2(2-1)} = n^0 = 1 \Rightarrow 1.Fall\ f(n) \in \Theta(1) \Rightarrow T(n) \in \Theta(logn)$

C) $T(n) = 2T(\frac{n}{2}) + n$

- $a = 2$
- $b = 2$
- $f(n) = n$

$n^{log_2(2)} = n^1 = n \Rightarrow 2.Fall\ f(n) \in \Theta(1) \Rightarrow T(n) \in \Theta(logn)$

3.3 Rekursionen aus alten Klausuren und geometrische Summenformel

A) Zeigen Sie, dass für folgende Rekursion $T(n) = \Theta(n^2 log n)$ ist.

$$T(1) = 0$$

$$T(n) = T(n-1) + nlogn$$

B) Sei $n = (\frac{8}{7})^k$ für ein $k \epsilon \mathbb{N}$. Folgende Rekursion ist für die Funktion T gegeben:

$$T(1) = 0$$

$$T(n) = \frac{7}{8} T(\frac{7}{8} n) + \frac{7}{8} n$$

Finden Sie für $T(n)$ eine geschlossene Form ohne das Mastertheorem zu verwenden und beweisen Sie die Korrektheit Ihrer geschlossenen Form mit vollständiger Induktion.

C) Sei $n = (\frac{3}{2})^k$ mit $k \epsilon \mathbb{N}$. Folgende Rekursion ist für die Funktion T gegeben:

$$T(1) = 0$$

$$T(n) = 2T(\frac{2}{3} n) + 1$$

Finden Sie für $T(n)$ eine geschlossene Form ohne das Mastertheorem zu verwenden und beweisen Sie die Korrektheit Ihrer geschlossenen Form mit vollständiger Induktion.

D) Sei n eine Zweierpotenz, das heißt $n = 2^k$ für ein $k \epsilon \mathbb{N}$. Folgende Rekursion ist für die Funktion T gegeben: Für $n > 1$ gelte

$$T(n) = A(n) + B(n),$$

wobei

$$A(n) = A(\frac{n}{2}) + B(\frac{n}{2})$$

und

$$B(n) = B(n-1)$$

+2n-1.

Die Endwerte seien $T(1) = 1$, $B(1) = 1$ und $A(1) = 0$. Finden Sie für $T(n)$ eine geschlossene Form ohne das Mastertheorem zu verwenden und beweisen Sie die Korrektheit Ihrer Lösung.

4 Blatt 02

4.1 Ternäre Suche

Bei der binären Suche ist der Input ein sortiertes Array A und eine Zahl x (die nicht notwendig im Array A vorkommen muss). Dabei wird A in zwei gleich große Teile A_1 und A_2 geteilt und ermittelt, in welchem der beiden Teile sich x befinden müsste. Dieses Verfahren wird *rekursiv* fortgesetzt. Betrachten Sie nun die *ternäre Suche*, bei der A statt in zwei Teile, in *drei* etwa gleich große Teile A_1, A_2 und A_3 geteilt wird.

A) Geben Sie ein Array A und ein zu suchendes Element x an, so dass die binäre Suche mit weniger Vergleichen auskommt als die ternäre Suche.

$A :=$

0	1	2	3	4	5	6

ges: **3**

binäre Suche:
$\lceil \frac{6-0}{2} \rceil + 0 = 3$. Position wird betrachtet
→ **3** gefunden

ternäre Suche:
$\lceil \frac{6-0}{3} \rceil + 0$ und $\lceil \frac{6-0}{3} \rceil + (\lceil \frac{6-0}{3} \rceil + 0)$, also Position 2 und 4 betrachtet
→ **3** nicht gefunden
→ neues Sucharray von Position 3 bis 3
→ Array Größe 1
→ **3** wird gefunden

Bei der binären Suche wird **3** also mit dem ersten Vergleich gefunden, bei der tertiären Suche jedoch erst nach dem drittem.

B) Geben Sie ein Array A und ein zu suchendes Element x an, so dass die ternäre Suche mit weniger Vergleichen auskommt als die binäre Suche.

$A :=$

0	1	2	3	4	5	6

ges: **2**

binäre Suche:
$\lceil \frac{6-0}{2} \rceil + 0 = 3$. Position wird betrachtet
→ Sucharray bis Position 3 betrachten: $\lceil \frac{3-0}{2} \rceil + 0 = \lceil 1,5 \rceil = 2$
→ Position 2 betrachten
→ **2** gefunden

→ 2 Vergleiche

ternäre Suche:
$\lceil \frac{6-0}{3} \rceil + 0 = \lceil 2 \rceil$
→ Position 2 betrachten
→ **2** gefunden
→ ein Vergleich

C) Geben Sie Pseudocode für die ternäre Suche an. Verwenden Sie dabei Rekursion. Numerieren Sie die Zeilen in Ihrem Pseudocode und erklären Sie detailliert jede Zeile Ihres Codes.

[H] sortiertes Array A, Integer x Integer [Postion von x in A] $oben \leftarrow length(1) - 1$
$unten \leftarrow 0$
while $(oben \geq unten)$
$a \leftarrow \lceil \frac{oben - unten}{3} \rceil$
if $(A(a + unten) > x)$ do
$oben \leftarrow a + unten - 1$
else if $(A(2a + unten) < x)$ do
$unten \leftarrow 2a + unten + 1$
else if $(A(2a + unten) > x)$ do
$unten \leftarrow a + unten + 1$
$oben \leftarrow 2a + unten - 1$
else if $(A(a + unten) == x)$ do
$(a + unten)$
else -1
end while
Erklärung:
Betrachte Array A
Falls $x \in A_1$,so prüft das der if-case in Zeile 5
Falls $x \in A_3$,so prüft das der if-case in Zeile 7
Falls $x \in A_2$,so prüft das der if-case in Zeile 9
Falls $x = A$ oder $x = 2a$, so prüfet das die if-cases in Zeile 12 und 14
Falls $x \notin A_1, A_2, A_3$ und $x \neq a, x \neq 2a$ so kann x nicht im geordneten Array A existieren.

D) Analysieren Sie die Zeitkomplexität der ternären Suche. Was können Sie über die asymptotische Laufzeit der ternären Suche im Vergleich zur binären Suche sagen?

$T(n) = T(\frac{n}{3}) + c$ mit $c \in 1,2,3,4,5$ ist die passende Rekursionsgleichung, da der Suchbereich nach jedem Rekursionsschritt nur noch $\frac{1}{3}$ des vorherigen Suchbereiches groß ist. c ist eine

Konstante (für die Vergleiche pro Rekursionsschritt) und ist immer 1,2 oder 3, außer das Element wird gefunden bzw. existiert nicht.
Im Mittel wird $x \approx 2$ sein.
Mit dem Master-Theorem:
$n^{log_3(1)} = n^0 = 1 \in c$
$\Rightarrow T(n) \in \Theta(n^{log_3(1)} log(n))$
$T(n) \in \Theta(log(n))$
Die binäre und ternäre Suche sind also asymptotisch gleich. Es gibt also keine signifikanten Laufzeitunterschiede zur binären Suche.

E) Bei jedem Rekursionsschritt werden ein oder zwei Vergleiche benötigt, um zu entscheiden, in welchem Teil des Arrays A das Element x liegt. Was ist die minimale, die durchschnittliche und die maximale Anzahl an Vergleichen die benötigt wird, wenn x nicht in A liegt.

Nehme an Array A hat Größen und $x \notin A$.
Minimale Anzahl von Vergleiche (best case): 5 Vergleiche, falls $x \notin A_1, A_2, A_3$ und $x \neq a, x \neq 2a$ im ersten Schritt.
Maximale Anzahl (worst case): $\frac{n}{3} * 3^n + 2 = n * 3^{n-1} + 2$, falls bis Arraygröße $n = 1$ aufgelöst wird und dann erst festgestellt wird $x \notin A$.
Durchschnittliche Anzahl (average case): $\frac{n * 3^{n-1} + 2 + 5}{2} = \frac{n}{2} * 3^{n-1} + 3,5$

4.2 Eine Anwendung der Binären Suche

Sei A ein sortiertes Array der Größe n und sei z eine gegebene Zahl. Das Ziel dieser Übung ist es, folgdende Frage zu beantworten: Gibt es in A zwei verschiedene Elemente x und y, so dass $x + y = z$?

A) Es sollte nicht schwierig sein, einen Algorithmus zu finden, der diese Frage in quadratischer Zeit (n^2) beantwortet. Geben Sie Pseudocode für einen solchen Algorithmus an und erklären Sie warum Ihr Algorithmus die Laufzeit (n^2) hat.

[H] sortiertes Array A, Integer z boolean[ob $x, y \in A$ mit $x + y = z$] $counter \leftarrow 0$
$counter2 \leftarrow 0$
while ($counter2 \leq length(A) - 1$)
while ($counter \leq length(A) - 1$)
if ($A(counter) + A(counter2) == z$)
if ($counter$ =!$counter2$)
$true$
else counter++
end while
counter2++
end while
$false$

Der Algorithmus hat $\mathcal{O}(z)$ Laufzeit, weil hier jede Zelle des Arrays einmal mit jeder anderen addiert und anschließendauf Größe = z verglichen wird.

B) Verwenden Sie nun die binäre Suche, um einen effizienteren Algorithmus zu finden, der die obige Frage in einer Laufzeit von $(n \log n)$ beantworten kann. Geben Sie auch hier Pseudocode an und begründen Sie die Korrektheit Ihres Algorithmus. Erklären Sie, warum Ihr Algorithmus die angegebene Laufzeit hat.

[H] sortiertes Array A, Integer z $counter \leftarrow 0$
$oben \leftarrow length(A) - 1$
$unten \leftarrow 0$
while $(counter \leq length(A) - 1)$
$y = z - A(counter)$
if$(binarySearch(y) > 0)$
$true$
end while
$false$

Da nun das Komplementärelement von y (also x mit $x + y = z$) nicht mehr in linearer Suchzeit, sondern logarithmischer Suchzeit $log(n)$ gefunden wird, falls vorhanden, benötigt dieser Algorithmus für jede Rekursion nicht mehr $\mathcal{O}(n)$ sondern $\updownarrow\}(\backslash)$ Schritte, was bei Tests für alle n Elemente des Input Arrays zu maximal $n * log(n)$ Schritten führt, was bedeutet, dass die Laufzeit in $\mathcal{O}(n * log(n))$ liegt.

C) Es ist klar, dass ein Algorithmus für die obige Frage mindestens die Laufzeit $\Omega(n)$ benötigt. Versuchen Sie, einen Algorithmus zu finden, der obige Frage in einer Laufzeit von (n) beantwortet. Geben Sie Pseudocode an und begründen Sie die Laufzeit und die Korrektheit Ihres Algorithmus.

[H] sortiertes Array A, Integer z boolean[ob $x, y \in A$ mit $x + y = z$] $counter \leftarrow 0$
$oben \leftarrow length(A) - 1$
while $(counter \leq length(A) - 1)$
$y = z - A(counter)$
if$(binarySearch(A, y;$ ab Counter +1 Index$) > 0)$
$true$
end while
$false$

Weil nun die Größe des binär durchsuchten Arrays in jedem Rekursionsschritt um 1 sinkt haben wir nicht mehr $log(n)$ Aufwand pro Rekursionsschritt, sondern $log(n - i)$ für das i-ten Rekursionsschritt. Das Array A(ab Counter +1) ist das Array ab der Position 5 (als untere Grenze für die binäre Suche) oder das Array ab Position 5 kopiert und als neues Array in die binarySearch mit gesuchtem y als Input gegeben. Das darf gemacht werden, weil ja die schon

abgeleiteten A(Counter) sicher kein Komplementärelement haben und dementsprechend auch keines sein können und somit bei der Suche ignoriert werden.

4.3 Implementierung von Suchalgorithmen

Laden Sie die Java-Vorlage aus dem Moodle herunter und implementieren Sie die folgenden Methoden:

1. die Lineare Suche in `linearSearch(int[] array, int key)`.
2. die Binäre Suche in `binarySearch(int[] array, int key)`.
3. die Interpolationssuche in `interpolationSearch(int[] array, int key)`.

Für die Implementierung der Interpolationssuche benutzen Sie die folgende Variante aus der Vorlesung, um das jeweils nächste Element zu bestimmen:

$$next \leftarrow \left\lceil \frac{a - S[unten-1]}{S[oben+1] - S[unten-1]} \cdot (oben - unten + 1) \right\rceil + (unten - 1)$$

siehe Main.java

5 Blatt 03

5.1 Sortieren

Betrachten Sie die Algorithmen `Insertionsort` und `Minimumsuche + Austausch` (Schematisch erklärt in Foliensatz 4, Seite 2).

A) Geben Sie jeweils Pseudocode für `Insertionsort` und `Minimumsuche + Austausch` an.

Insertionsort:

```
Input: Array A
Output: Sortiertes Array S

n = length(A) - 1
g = 0

while(g < n) do
    ticker = g;
    x = A[g+1];
    while(A[g+1] <= A[ticker]) do
        A[ticker + 1] = A[ticker];
        if (ticker == 0) do
            brake;
        end if
        ticker --;
    end while
    A[ticker + 1] = x;
    g++;
 end while

 return A;
```

Minimumssuche + Austausch:

```
Input: Array A
Output: Sortiertes Array S

n = length(A) - 1
i = 0

while (i < n) do
    min = i;
    j = i + 1;
```

```
    while(j <= n) do
        if(A[min]>A[j]) do
            min =j;
        end if
        j++;
    end while
    x = A[i];
    A[i]= A[min];
    A[min] = x;
    i++;
end while

return A;
```

B) Argumentieren Sie, dass beide Sortierverfahren korrekt sortieren.

Insertion-Sort:
Bei Insertion Sort wird von einer aufsteigenden Grenze g bis n = length(A) - 1 immer abwärts verglichen ob das Element direkt nach g (also g+1) kleiner als einer seiner Vorgänger ist, bis dieses es schließlich nicht mehr ist.
Dabei wird, bevor A[g+1] verglichen wird, dieses der Inhalt dieses Arrayfeldes als x gespeichert um später an die passende Position eingesetzt werden zu können und nach jedem Vergleich bei dem A[g+1] kleiner ist das betrachtete Element eine Position weiter nach rechts geschoben (weswegen A[g+1] auch gespeichert werden musste).
Ganz grob/kurz gesagt:
- Insertion Sort geht von links nach rechts durch
- Grenze zunächst auf erstem Element (an Position 0)
- Grenze steigt nach jedem Schritt um 1
- bei jedem Schritt wird das Element direkt nach der Grenze mit jedem Element zuvor verglichen bis es größer ist und passen eingeordnet.

Minimumsuche + Austausch:
In Zeile 12 bis 17 wird das Minimum gesucht. In Zeile 18 bis 20 wird getauscht.
Hierbei tickt i von 0 bis n = length(A)-1 hoch und vertauscht bei jedem Schleifendurchgang das gefundene Minimum (welches immer ab Position i linear gesucht wird) mit dem Element an der i-ten Position.
Sobald i = n erreicht wird, ist das letzte Element automatisch richtig sortiert und daher wird schon bei while(i kleiner n) [Zeile 9] abgebrochen und anschließend das (sortierte) Array zurückgegeben.

C) Geben Sie die Anzahl der Vergleiche und die Anzahl der Vertauschungen auf einer vorsortierten Eingabe der Länge n an.

Insertion Sort:
Vergleiche: 1 (erstes A[0] mit zweitem A[1]) + ... + 1 (vorletztes A[n-1] mit letztem A[n]) = n - 1 Vegleiche.
Austausche: Theoretisch n - 1 mal, da das betrachete Element von A[g + 1] bei vor jedem g++ durch sich selbst überschrieben wird, wenn man das als Vertauschung zählt. Zählt man das "sich selbst austauschen/überschreiben" nicht, so sind es 0 Vertauschungen.

Minimumsuche + Austausch:
Vergleiche: Da bei der Minimumssuche jedes Element nach Position i im Array einmal komplett durchlaufen wird: n + (n - 1) + ... + 1 = $((n+1)*n)/2$ Vergleiche.
Austausche: Da nach jeder Minimumssuche genau einmal (mit sich selbst vertauscht/durch sich selbst überschieben) Vertauscht wird, wird bei gesamtem Durchlauf des Arrays, wie oben theoretisch n mal getauscht. Zählt man eine Selbstüberschreibung jedoch nicht als Tausch, so sind es 0 Austausche, da die Folge bereits sortiert ist und jedes Element bereits an seinem einzufügenden Platz nach der Minimumssuche saß/sitzt.

D) Konstruieren Sie für allgemeines $n \in \mathbb{N}$ je ein Beispiel, auf denen die Algorithmen eine maximale Anzahl von *Vergleichen* benötigt. Geben Sie diese Anzahl auch an.

Array mit $a_i > a_{i+1}$:

Insertion Sort:
Weil der Reihe nach vorgegangen wird und die Eingabe von groß nach klein sortiert ist, werden für jeden Schritt i, genau i Vergleiche benötigt, um die richtige Position des Elements zu bestimmen:$\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ Vergleiche.

Minimumsuche + Austausch:
Bei der Minimumssuche muss das Array immer vollständig durchlaufen werden. Es ergibt sich also $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ Vergleiche.

E) Konstruieren Sie für allgemeines $n \in \mathbb{N}$ je ein Beispiel, auf denen die Algorithmen eine maximale Anzahl von *Vertauschungen* benötigt. Geben Sie diese Anzahl auch an.

Array mit $a_i > a_{i+1}$:

Insertion Sort:
Weil der Reihe nach vorgegangen wird und die Eingabe von groß nach klein sortiert ist, werden für jedes g genau g Vertauschungen benötigt bis das Element sein passende Stelle gefunden hat und für jedes nächste g, das Element wieder ganz nach links geschoben wird.
Es ergeben sich $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ Vertauschungen.

Minimumsuche + Austausch:

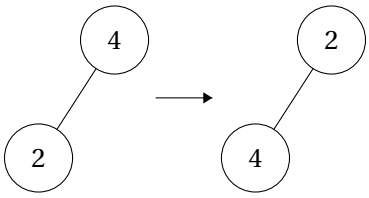
Die Anzahl der Vertauschungen liegt hier statisch bei n (außer man zählt nicht wenn ein Element mit sich selbst vertauscht wird und gerade dann wäre dies hier sicher n).

5.2 HEAPSORT

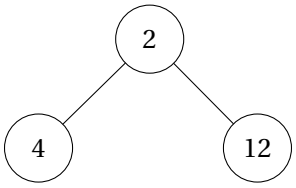
Gegeben sei das Array $A = \langle 4, 2, 12, 10, 18, 14, 6, 16, 8 \rangle$.

A) Bilden Sie schrittweise (Element für Element) den Min-Heap S für das Array A. Benutzen Sie dabei die Heap-Eigenschaft: Jeder Baumknoten u ist mit einem Element $S[u]$ beschriftet und es gilt: Ist u Elternknoten von v, so ist $S[u] \leq S[v]$. Veranschaulichen und kommentieren Sie alle Schritte.

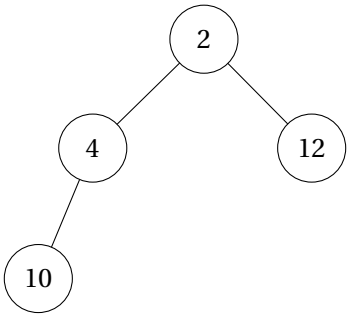
Schritt 1: 1. Array-Element wird in leeren Heap eingefügt.



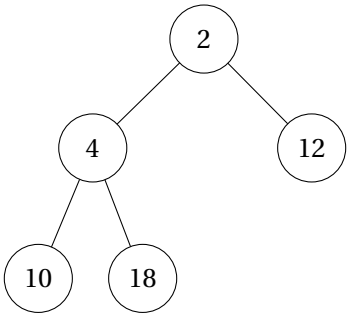
Schritt 2: 2. Array-Element wird in Heap eingefügt und Heap-Eingeschaft geprüft und hergestellt.



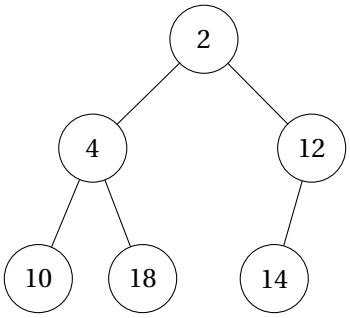
Schritt 3: 3. Array-Element wird in Heap eingefügt. Heap-Eigenschaft erfüllt, kein Austausch.



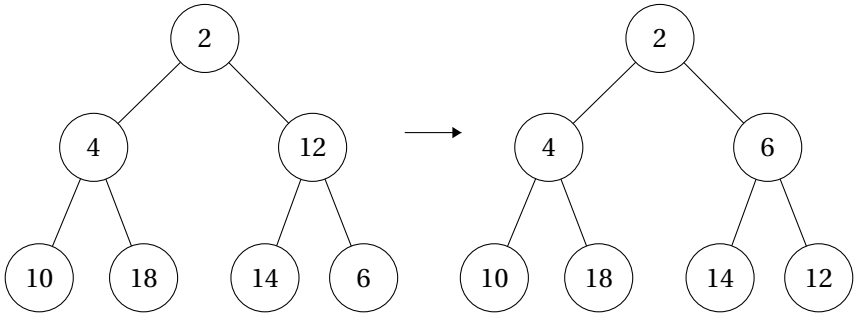
Schritt 4: 4. Array-Element in Heap einfügen. Heap-Eigenschaft erfüllt, kein Austausch.



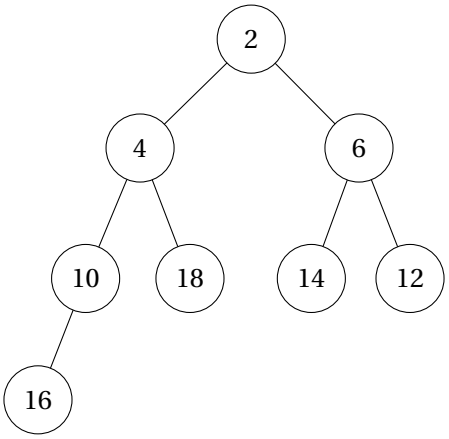
Schritt 5: 5. Array-Element wird in Heap eingefügt. Heap-Eigenschaft erfüllt, kein Austausch.



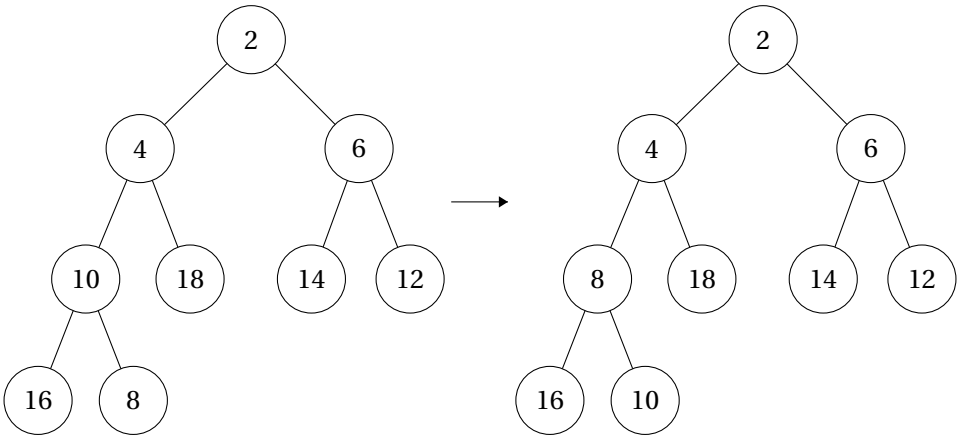
Schritt 6: 6. Array-Element wird eingefügt. Heap-Eigenschaft erfüllt, kein Austausch.



Schritt 7: 7. Array-Element wird in Heap eingefügt und Heap-Eigenschaft geprüft und hergestellt.



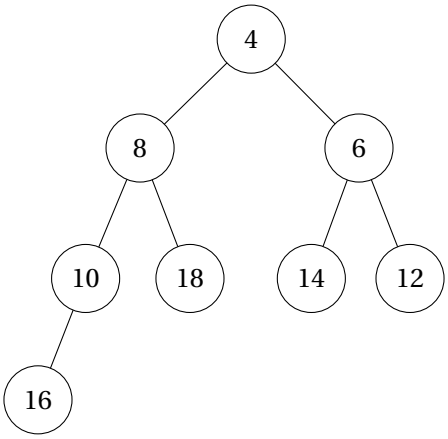
Schritt 8: 8. Array-Element wird in Heap eingefügt. Heap-Eigenschaft erfüllt, kein Austausch.



Schritt 9: 9. und letztes Array-Element wird in Heap einfügen und Heap-Eigenschaft wird geprüft und hergestellt. Min-Heap ist vollständig erstellt.

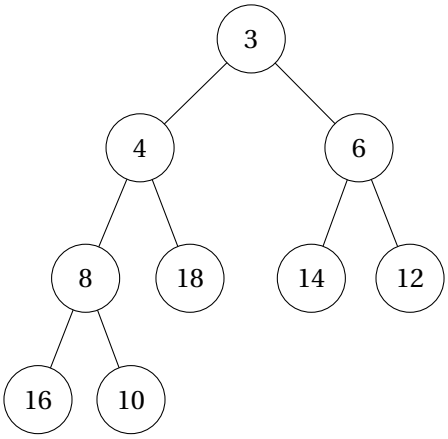
B) Wie sieht der Heap aus, wenn Sie eine EXTRACTMIN Operation ausgeführt und dann die Heapeigenschaft wieder hergestellt haben?

Nach ExtractMin:



Durch Extractmin wird die Wurzel entfernt, da Wurzel das kleinste Element ist.
Dann wird 10 zur Wurzel und sinkt nach unten.
Dabei wird erst 10 mit 4 getauscht und anschließend 10 mit 8 vertauscht.

C) Fügen Sie das neue Element 3 zu dem Heap (aus b) hinzu.



Das Element 3 wird am Ende des Arrays angehängt und anschließend die Heapeigenschaft getestet.
3 steigt dabei nach dem Vergleich mit der 10, 8 und 4 bis zur Wurzel auf.

D) Analysieren Sie in $\mathcal{O}$-Notation die Laufzeit der Methode EXTRACTMAX, die das maximale Element aus einem Min-Heap S der Größe n löscht.

ExtractMax benötigt n - 1 Schritte, um alle Blätter des Heaps zu finden und dann noch einmal maximal $2^{Hoehe(H)} + 1$ Schritte für den Vergleich der Blätter, das Finden und das Löschen des

Maximums.
Weil die Höhe des Heaps bei jeder nächsten Zweierpotenz um 1 steigt, haben wir $Hoehe(H) = \lfloor \log_2 n \rfloor$, also die Laufzeit $T(n) = n - 1 + 2^{Hoehe(H)} + 1 \leq n + 2^{\log_2 n} = n + n = 2n \Rightarrow \mathcal{O}(n)$.

5.3 Eine Erweiterung von Heapsort

In der Vorlesung haben Sie das Sortierverfahren Heapsort kennengelernt. Wir betrachten nun eine Erweiterung dieses Verfahrens, das sogenannte *k-Heapsort*. Dabei ist $k \geq 2$ eine natürliche Zahl. Bei diesem Verfahren benutzt man statt einem Binärbaum einen k-nären Baum, bei dem jeder Knoten höchstens k Kinder hat. Deshalb heißt der korrespondierende Heap k-Heap.

A) Wie kann man einen k-Heap als ein Array repräsentieren? Wie effizient ist es, die Kinder bzw. den Elternknoten eines gegebenen Knoten zu finden?

Array A[a_0.....a_n]
Root: a_0
1st Level: a_1.....a_{2+k}
2nd Level: a_{1+k}.....a_{2+k+k^2}
.
.
.
nth Level: $a_{1+\sum_{i=1}^{n-1} k^i}$...$a_{\sum_{i=1}^{n} k^i}$

Parent nodes of i: $\lfloor (i-1)/k \rfloor$
Child nodes j of i: $i * k + j$ $(1 \leq j \leq k)$

B) Geben Sie die Höhe eines k-Heaps an, wenn dieser n Elemente enthält.

h = $\lceil log_k((n*k) - (n+1)) \rceil - 1$

C) Geben Sie Pseudocode für effiziente Implementierungen der Methoden `Insert` und `ExtractMin` an. Analysieren Sie die Komplexität der beiden Methoden in Abhängigkeit von n und k.

A = Array, X = Element, I = Index, k

```
Insert(A, X, k) {
    Increase length of A by 1
    Insert X at A[length(A)]
    HeapUp(A, length(A) - 1, k)
}

HeapUp(Array A, i, k) {
```

```
    while (i not root){

      if (array[i] > array[parent-position] {
          Swap A[i] and A[parent-position]
          i = parent
          parent = (i-1)/k
      }
      else
        break
        }
}
```

Complexity:
Because we need a comparison for each tree-level: Insert() $\in \mathcal{O}(log_k((n*k)-(n+1))$

```
ExtractMin(Array A)
{
        Extract root
    Set length(A)-1 (last element) as new root
    Shorten array by 1
    HeapDown(A, root)
}

HeapDown(array A, node n)
{
        for (n not leaf){
        compare n to its childs 1 - k step by step
                if(n < child){
                swap (n, child)
            }
         HeapDown(A, child)

    }
}
```

Complexity:
We need k comparisons on every level, thats why extractMin() $\in \mathcal{O}(3log_k(n*k-n)$.

5.4 Zwei-Drittel-Sortieren

Eine alternative Methode, um ein Array A der Länge n zu sortieren, ist die Folgende:
[H] $A[left] > A[right]$ exchange $A[left]$ and $A[right]$ $left+1 \geq right$ $k \longleftarrow \left\lfloor \frac{right-left+1}{3} \right\rfloor$
`ZweiDrittelSortieren`(A, $left$, $right-k$) `ZweiDrittelSortieren`(A, $left+k$, $right$)

`ZweiDrittelSortieren`(A, $left$, $right-k$) `ZweiDrittelSortieren`(A, $left$, $right$)

A) Argumentieren Sie, dass `ZweiDrittelSortieren`(A, 1, n) das Array $A[1..n]$ korrekt sortiert.

Argumentation, dass Zwei-Drittel-Sortieren korrekt ist, mithilfe von Induktion über die Länge des sortierten Teilarrays k: $(right - left + 1)$.
Für Länge 1 ist das Array immer sortiert, für Länge 2 muss einmal verglichen und gegebenenfalls ausgetauscht werden.

Für Längen > 2 sei $k = (right - left + 1)/3$.
Betrachte nun die k größten Elemente des Arrays von left bis right.
$m \leq k$ Elemente sind in den ersten zwei Dritteln des Arrays und gleichzeitig sind m die größten Elemente dieses Teilarrays.
Nach der ersten Rekursion, die das Teilarray korrekt sortiert hat (Induktionsvoraussetzung korrekt), befinden sich diese m Elemente im Teilarray A$[right - k - m + 1]$...A$[right - k]$ mit der Eigenschaft $right - k - m + 1 \geq left + k$.

Das mittlere Drittel besteht aus mindestens k Elementen und die k größten Elemente befinden sich zwischen $left + k$ und $right$, also sind diese auch nach dem zweiten Aufruf korrekt sortiert.
Im letzten Drittel des Arrays die (übrigens) ≤ k größten Elemente, welche im letzten rekursiven Aufruf sortiert werden.

B) Analysieren Sie die Laufzeit von `ZweiDrittelSortieren` im worst-case. Geben Sie Ihre Angaben in $\mathcal{O}$-Notation an.

Worst-Case: In jedem Rekursionsschritt werden zwei Elemente ($left$, $right$ des jeweiligen Teilarrays) getauscht werden und anschließend weiter gedrittelt, daher $T(n) = 3 * T(\frac{n}{3}) + 2$.

Mastertheorem:
$a = 3$, $b = 3$, $f(n) = 2 * n^0$
$f(n) = 2 * n^{1-1} = 2 * n^{\log_3(3)-1} \in \mathcal{O}(n^{\log_3(3)-\epsilon})$ wobei $\epsilon = 1$
$\Rightarrow T(n) \in \Theta(n^{\log_3(3)}) \in \Theta(n^1) \in \Theta(n)$

C) Ist Zwei-Drittel-Sortieren im worst-case effizienter als Insertsort, Minimumsuche+Austauschen, Quicksort oder Heapsort? Alle Antworten sollten jeweils ausreichend begründet werden.

Insertion Sort Worst-Case Laufzeit:
$T(n) = \frac{n^2+n}{2} = \mathcal{O}(n^2)$, zwei-Drittel-Sortieren ist also deutlich schneller.
Minimumsuche + Austausch hat eine Worst-Case Laufzeit von $T(n) = \frac{n^2+n}{2} = \mathcal{O}(n^2)$, zwei-Drittel-Sortieren ist also deutlich schneller.
Quicksort hat die Worst-Case Laufzeit $\mathcal{O}(n^2)$, zwei-Drittel-Sortieren ist also deutlich schneller.
Heapsort hat die Worst-Case Laufzeit $\mathcal{O}(n \log n)$, zwei-Drittel-Sortieren ist also etwas schneller.

6 Blatt 05

6.1 Bucketsort

Gegeben seien n Zahlen im Bereich $[0,\ldots,n^k-1]$ für ein festes $k \in \mathbb{N}$. Zeigen Sie, dass mittels Bucketsort die Zahlen mit einer Worstcase-Laufzeit von $O(n)$ sortiert werden können.
Hinweis: Stellen Sie die Zahlen zur Basis n dar.

Bucketsort sortiert jede Zahl im ersten Schritt an der letzten Stelle in das passende von 10 Buckets.
Für k = 1, 2, ... steigen die Längen der Zahlen für jeweiliges n>1 grob um 2*length($n^{k-1}-1$).
Also wird für höheres k häufiger in Buckets einsortiert wegen der Stellenlänge der höchsten Zahl im Array, genau der Stellenanzahl n^k-1.
Die Länge hierzu lässt sich mit $\prod_{i=Stellenlaengevonn^k-1}^{k} i$ bei festem k bestimmen, was $Stellenlaenge*k!$ ist.
Wir haben also $10*Stellenlaenge*k!*n$ Suchläufe und da k fest ist, sowie auch die Stellenlänge kann dies als Konstante c dargestellt werden: $c*n \in \mathcal{O}(n)$.

6.2 Mediansuche in Linearzeit

A) In der Vorlesung haben Sie gelernt, wie Sie den Median von fünf Zahlen mit 7 Vergleichen bestimmen können. Verallgemeinern Sie dieses (rekursive) Vorgehen für Eingaben ungerader Länge. Geben Sie die Anzahl benötigter Vergleiche für Listen der Länge n, mit $n \in \{7,9,11,13,15,17\}$ an.

1. Aufteilen der Menge in Paare → am Ende soll eine Zahl übrig bleiben. Die Zahlenpaare werden miteinander verglichen ($\lfloor\frac{n}{2}\rfloor$ Vergleiche)
2. Bestimme das Minimum unter allen Zahlen, die im Zahlenpaar kleiner waren im Vergleich($\lfloor\frac{n}{2}\rfloor-1$ Vergleiche)
3. das gleiche für das Maximum ($\lfloor\frac{n}{2}\rfloor-1$ Vergleiche)
4. Minimum und Maximum aus dem Array nehmen und den Algorithmus auf das Array Größe n-2 aufrufen (V(n-2) Vergleiche)

Es folgt: $V(n)=\lfloor\frac{n}{2}\rfloor+2*(\lfloor\frac{n}{2}\rfloor-1)+V(n-2)$

Rekursionsende: V(5)=7

Daraus folgt folgende Anzahl an Vergleichen für Listen der Länge n:

n	V(n)
7	$\lfloor\frac{7}{2}\rfloor + 2 * (\lfloor\frac{7}{2}\rfloor - 1) + V(5) = 14$
9	$\lfloor\frac{9}{2}\rfloor + 2 * (\lfloor\frac{9}{2}\rfloor - 1) + V(7) = 24$
11	$\lfloor\frac{11}{2}\rfloor + 2 * (\lfloor\frac{11}{2}\rfloor - 1) + V(9) = 37$
13	$\lfloor\frac{13}{2}\rfloor + 2 * (\lfloor\frac{13}{2}\rfloor - 1) + V(11) = 53$
15	$\lfloor\frac{15}{2}\rfloor + 2 * (\lfloor\frac{15}{2}\rfloor - 1) + V(13) = 72$
17	$\lfloor\frac{17}{2}\rfloor + 2 * (\lfloor\frac{17}{2}\rfloor - 1) + V(15) = 94$

B) Für die Laufzeitanalyse der Medianbestimmung ergab sich bei einer Unterteilung in Fünfergruppen die folgende Rekursionsgleichung:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + c \cdot n$$

I) Geben Sie eine möglichst gute Konstante c an.

z.z. $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10} * n) + c_1 * n$

in Vorlesung bewiesen, dass $T(n) \leq c_2 * \frac{n}{5} + c_2 * \frac{7n}{10}$ ist; $d = \frac{c_2}{10}$

damit gilt: $d = c_1$

somit ist $c_2 * \frac{n}{5} + c_2 * \frac{7n}{10} + d * n = \frac{n}{5} + \frac{7n}{10} + c_1 * n$

$\frac{n}{5} + \frac{7n}{10} + c_1 * n$

$= \frac{9n}{10} + d * n$

$\Rightarrow c_2 = 1$

$\Rightarrow d = \frac{1}{10}$

$\Rightarrow c_1 = \frac{1}{10}$

II) Wie verändert sich die Rekursionsgleichung, wenn die Liste in Siebener- statt in Fünfergruppen geteilt wird? Geben Sie auch hier eine möglichst gute Abschätzung der Konstante c an.

z.z. $T(n) \leq T(\frac{n}{7}) + T(\frac{10}{14} * n) + c_1 * n$

dieselbe Berechnung wie oben

$\frac{n}{7} + \frac{10n}{14} + d * n$

$= \frac{12n}{14} + d * n$

$\Rightarrow c_2 = 2$

$\Rightarrow d = \frac{2}{14}$

$\Rightarrow c_1 = \frac{2}{14}$

6.3 Median in sortierten Arrays

Seien $A[1..n]$ und $B[1..n]$ zwei sorrtierte Arrays der Größe n.

A) Beschreiben Sie einen Algorithmus, der den Median aller $2n$ Elemente aus A und B in Zeit $O(\log n)$ findet.

[H] **Data** Array A, Array B (beide mit Länge n)
Result Median m aller 2n Elemente aus A und B
make new Array C mit Größe 2n
int k = 0 (Position in A)
int l = 0 (Position in B)
for 0 to 2n-1 do
if A[k] ≤ B[l]
C[i] ← A[k]
k++
else
C[i] ← B[l]
l++
end if
end for
int $d = \frac{2n}{2} - 1$
int $m = \frac{C[d] + C[d+1]}{2}$
return m

B) Begründen Sie die Laufzeit und die Korrektheit Ihres Algorithmus.

Laufzeit O(n)

6.4 KONVEXE HÜLLE

Gegeben sei eine Menge $M = \{(x_i, y_i) \in \mathbb{Q}^2 \mid 1 \leq i \leq n\}$. Die konvexe Hülle conv(M) einer Menge M ist der kleinste Polyeder der alle Punkte in M enthält. Eine einfache Repräsentation der konvexen Hülle ist die kleinste Menge $N \subseteq M$ von Punkten, so dass conv(N) = conv(M). Bildlich gesprochen: die *Eckpunkte* der Menge M. Eine *zyklische Sortierung* der Eckpunkte ist eine Ordnung der Punkte in N so, dass beim Ablaufen der Punkte nach der Sortierung ein Weg um das Polygon entsteht.

Nehmen Sie an, dass Sie einen Algorithmus haben der in Zeit $T(n)$, gegeben eine Menge M (wie oben), eine zyklische Sortierung der Eckpunkte der konvexen Hülle ausgibt. Zeigen Sie, dass $\Omega(n \log n)$ eine untere Schranke für die worst-case Laufzeit des Algorithmus ist.

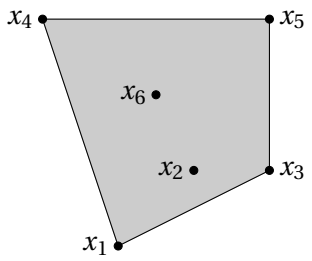


Figure 6.1: Der Algorithmus erhält die Punktemenge $\{x_1, \ldots, x_6\}$ und gibt die Punkteliste (x_1, x_4, x_5, x_3) aus. Der grau gefärbte Bereich ist die konvexe Hülle.

geg:

- Algorithmus in Zeit T(n)
- Menge $M = (x_i, y_i) \in \mathbb{Q}^2 | 1 \leq i \leq n$
- zyklische Sortierung der Eckpunkte der konvexen Hülle

z.z. Ω (n log n) ist eine untere Schranke für worst-case Laufzeit des Algorithmus

Probleme:

1. konvexe Hülle bestimmen
2. Verwendung des vergleichsbasierenden Sortierens; läuft also bestenfalls auch n log n

Problemlösung:

- transformieren einer Instanz I(B) (ein Array X) zu einer Instanz I(A)
  generieren zu jeder Zahl aus X einen Punkt (x_i, x_i^2)
  läuft in Zeit O(n) → einmaliges Durchlaufen des Arrays und währenddessen das Erzeugen des jeweiligen Punktes

- man erhält n Punkte, welche alle auf einer Parabel liegen
  die konvexe Hülle besteht dann genau aus diesen Punkten
- Algorithmus soll Instanz I(A)lösen
  in angegebener Zeit T(n)
- Lösung soll nun zu einer Lösung von Instanz I(B) transformiert werden
  soll eine sortierte Liste werden
  zuerst wird der Eckpunkt mit der kleinsten x-Koordinate ausgewählt
  x- Koordinate dieses Punktes kommt an die erste Stelle des Ergebnisarrays
  alle folgenden Werte werden von der Parabel (von links nach rechts) abgelesen

6.5 Untere Schranke

Sei A eine sortierte Liste der Größe n.

A) Das erste Problem besteht daraus, zu entscheiden, ob in der Liste A Duplikate vorkommen. Beweisen Sie, dass $n-1$ eine untere Schranke für die Anzahl der Vergleiche ist, die man dafür braucht.

3 Fälle:

1. Fall
   Abgleichen des ersten Elementes mit der Liste und finden des erstes Elementes an der letzten Stelle in der Liste. Dann muss man n-1 mal vergleichen, also das Erste mit jedem Element der Liste, außer mit sich selbst.
2. Fall
   Abgleichen des ersten Elementes mit der Liste und finden des erstes Elementes an der letzten Stelle in der Liste. Hierbei muss man n-(n-Anzahl Stellen vor Listenende) vergleichen. Da aber $\Omega(n-(n-1)) \in \Omega(n-1)$, auch hier untere Grenze von $\Omega(n-1)$
3. Fall

B) Das zweite Problem ist es, zu entscheiden, ob in der Liste A ein beliebiger Wert x vorkommt. Beweisen Sie, dass im worst-case $\Omega(\log n)$ Vergleiche dafür benötigt werden (unabhängig vom verwendeten Algorithmus).

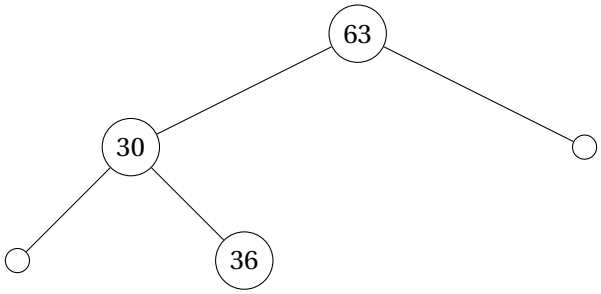
7 BLATT 06

7.1 AVL-BÄUME

A) Fügen Sie die Zahlen 63, 30, 36, 31, 12, 50, 35, 5, 27, 59, 43, 17 (in dieser Reihenfolge) in einen (zu Beginn leeren) AVL-Baum ein. Sie dürfen die Schritte, in denen nicht rotiert, sondern nur eingefügt wird, zusammenfassen.

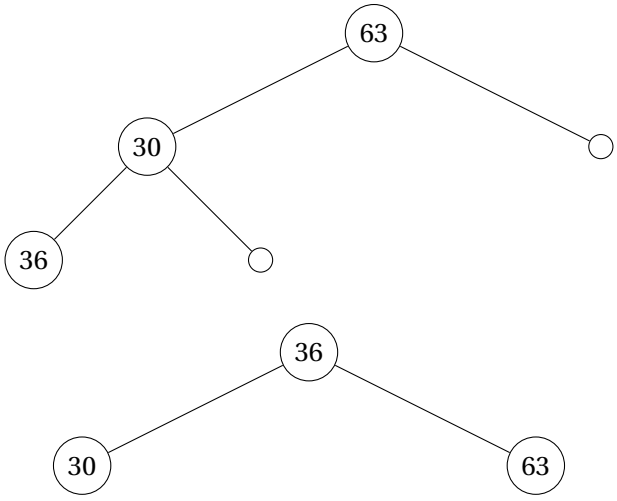
Schritt 1

Schritt 2,3

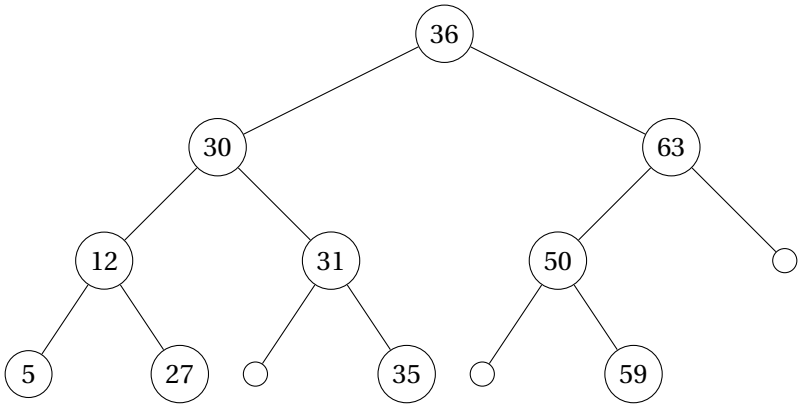


$bal(63) = -2, bal(30) = 1$
$\Rightarrow$ VZ ungleich
$\Rightarrow$ Doppelrotation

Schritt 4

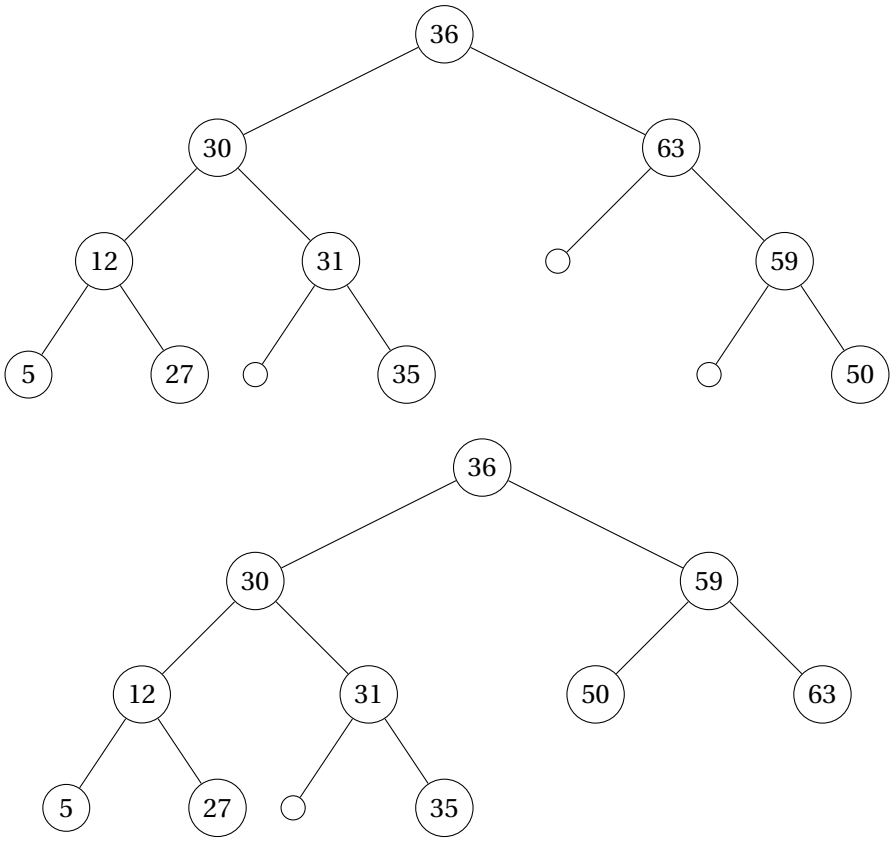


Schritt 5,6,7,8,9,10,11

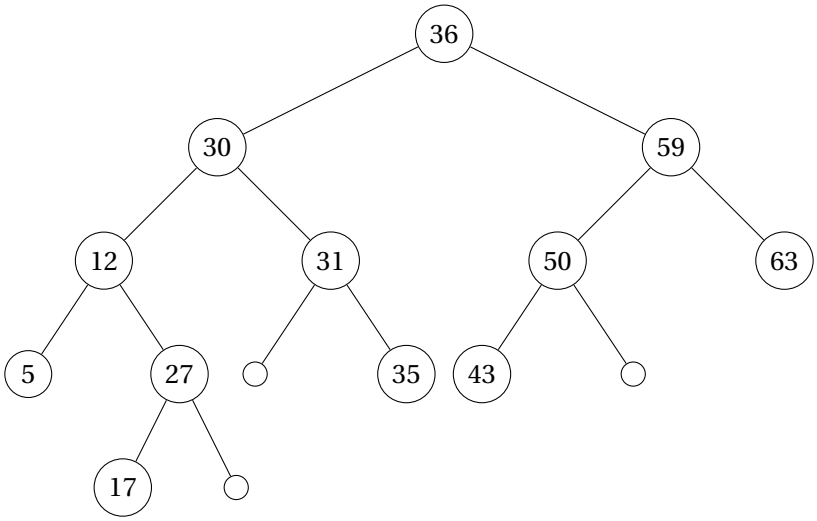


$bal(63) = -2, bal(50) = 1$
$\Rightarrow$ VZ ungleich
$\Rightarrow$ Doppelrotation

Schritt 12



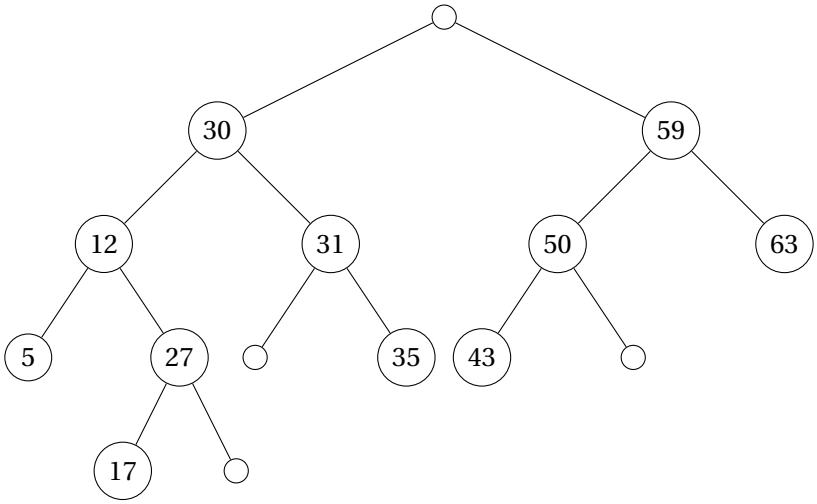
Schritt 13,14



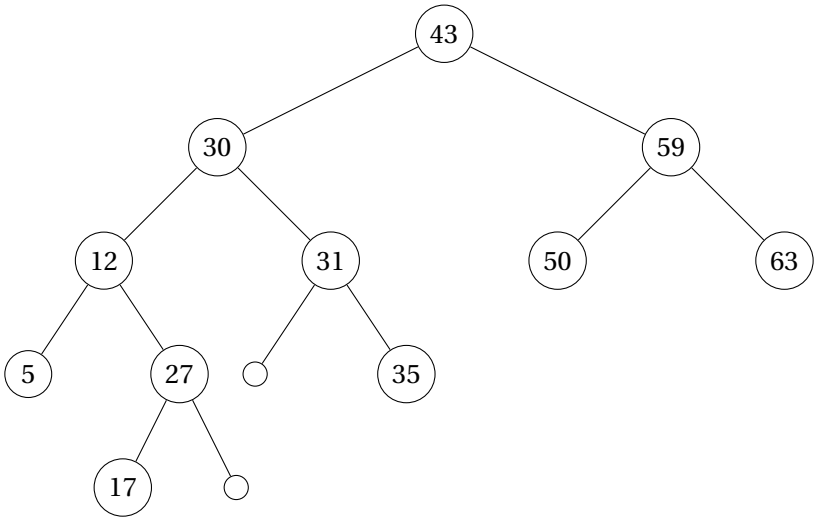
$|bal(u) \leq 1| : \forall n$, n Knoten $\Rightarrow$ fertig

B) Löschen Sie die Zahlen 36, 43 wieder aus dem Baum. Geben Sie die Schritte an.

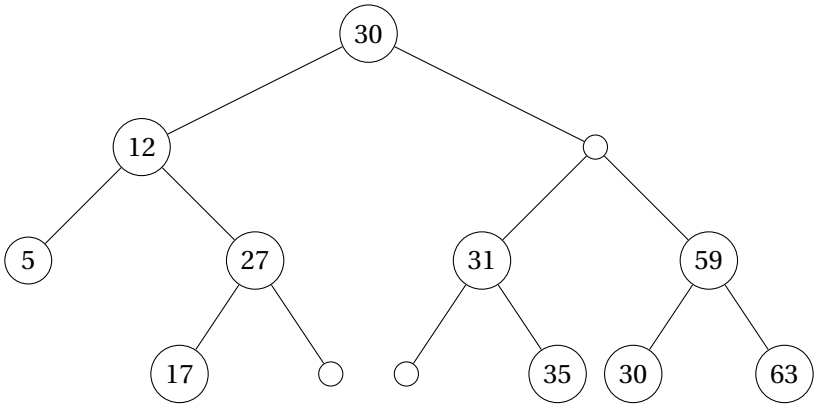
Lösche 36 und hefte 43 an Baum



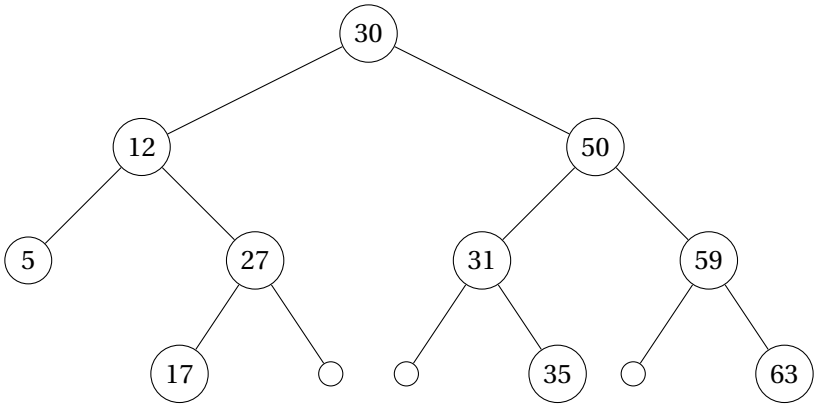
nachdem 36 gelöscht



Rotation



nach 43 gelöscht



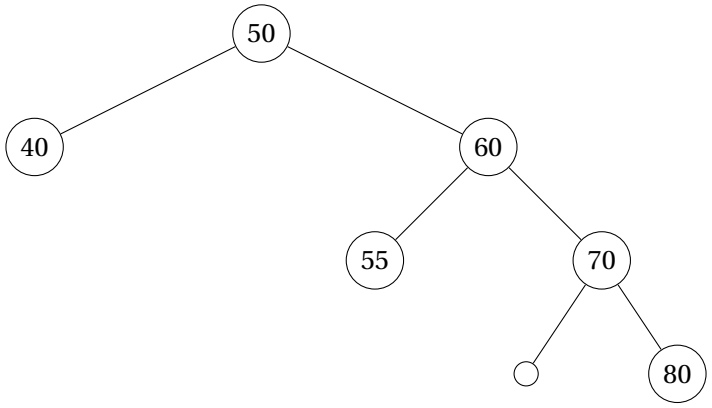
C) Leiten Sie eine Eingabefolge von n Zahlen für einen AVL-Baum ab, die keine Rotationen erfordert.

Nehme an die Eingabe ist oder wurde zuvor sortiert. Wähle nun als Einfüge-Element den Median und teile die Eingabe in zwei weitere sortierte Eingaben. Wähle nun als Einfüge-Element zuerst den linken, dann den rechten Median. Nun gibt es vier Eingaben (sortiert) von denen wieder zuerst die Mediane bestimmt werden müssen und ausschließend nacheinander eingefügt werden.
Die Eingabefolge ergibt sich nur aus der Einfüge-Reihenfolge der Elemente der sortierten Eingabe: Sortiere eine sortierte Eingabefolge um, dass die Mediane der sortierten Teilfolge von links nach rechts aufgereiht werden. Falls die Eingabefolge keine eindeutigen Mediane hat, füge beide Median-Element hinzu.

7.2 Binäre Suchbäume

A) Gegeben ein Pfad P von der Wurzel zum Blatt eines binären Suchbaumes. Sei B die Menge der Knoten auf diesem Pfad, A alle Knoten links von P und C alle Knoten rechts von P. Gilt dann $a \leq b \leq c$ für alle $a \in A$, $b \in B$ und $c \in C$?

Nein Gegenbeispiel:



ist binärer Suchbaum, weil ∀ Knoten n gilt:

- $childr(n) \geq n$
- $childl(n) < n$

Wähle Pfad $P = (50, 60, 70, 80)$
$\Rightarrow A = (40, 55); C = \emptyset$
$|P| = 4, |A| = 2, |C| = 0$
$|A| \leq |P| > |B| \neq |A| \leq |P| \leq C$
$a \leq b \leq c$ mit $a \in A, b \in B, c \in C$ gilt nicht

B) Zeigen Sie: Hat ein Knoten in einem binären Suchbaum zwei Kinder, so hat sein Nachfolger kein linkes Kind und sein Vorgänger kein rechtes Kind. Gilt diese Aussage auch für ein Knoten mit nur einem Kind?

Sei $x_i \in \mathbb{Z}$ und $x_0 < x_1 < ... < x_n$

Geg:

- x_i 2 Kinder
- x_{i-1} Vorgänger von x_i
- x_{i+1} Nachfolger

Beh: x_{i-1} hat kein rechtes Kind, x_{i+1} hat kein linkes Kind

Bew Da x_{i-1} Vorgänger von x_i:

$\Rightarrow x_{i-1} < x_i \wedge \nexists x_j \in [x_{i-1}; x_i]$

Da nun x_{i-1} nicht Kind von x_i (weil x_i schon 2 Kinder hat) sein kann, existiert kein x_j zwischen x_{i-1} und x_i

$\Rightarrow x_{i-1}$ Vorgänger kann kein rechtes Kind x_j mehr erhalten

Da x_{i+1} Nachfolger von x_i:

$\Rightarrow x_{i+1} > x_i \wedge \nexists x_j \in [x_i; x_{i+1}]$

Da x_i schon 2 Kinder hat

$\Rightarrow x_{i+1}$ Nachfolger kann kein rechtes Kind x_j mehr erhalten

Diese Aussage gilt für Knoten mit nur einem Kind, aber nur falls Knoten bereits linkes Kind besitzt bei Vorgänger-Einfügung. Ebenso gilt die Aussage falls Knoten bereits rechtes Kind besitzt für den Nachfolger.

C) Zeigen oder widerlegen Sie: Sei T ein binärer Suchbaum und seien x und y Elemente in T. Der binäre Suchbaum, der sich nach Löschen von x und y (in dieser Reihenfolge) ergibt ist derselbe Suchbaum, der sich nach Löschen von y und x (in dieser Reihenfolge) ergibt.

T Bin. SB, $x, y \in binSB$

Da beim Löschen immer das kleinste Element des rechte oder das größte des linken TB gewählt werden, wird immer ein Blatt für x oder y als Ersatz gewählt, welches anschließend

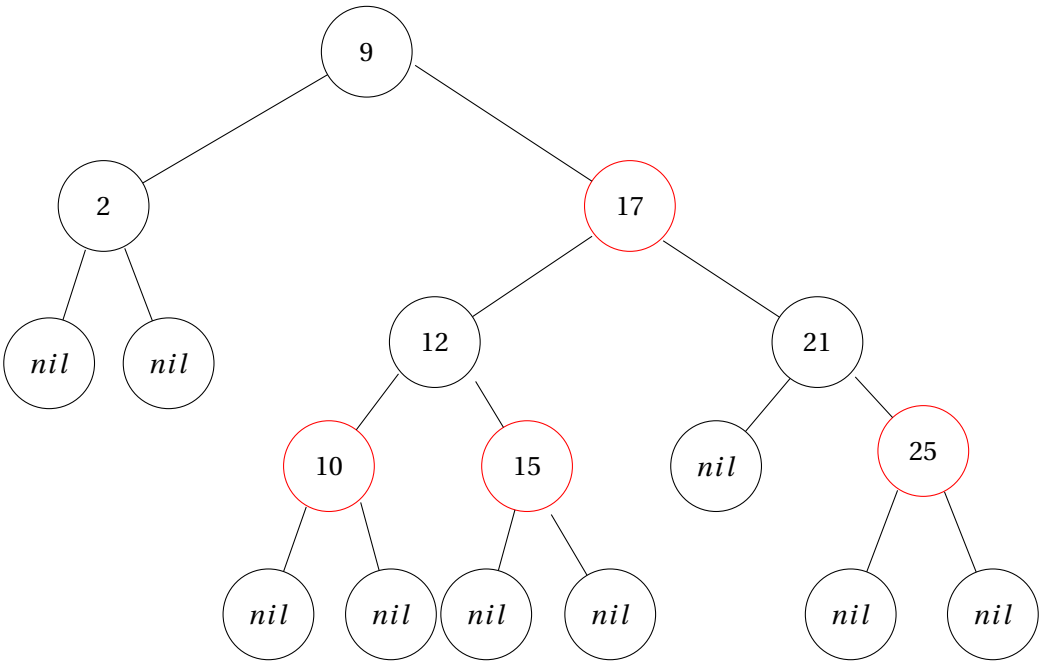
gelöscht wird. Da sich dadurch die Reihenfolge innerhalb des Baumes dadruch nicht verändert kann in beliebiger Reihenfolge gelöscht werden.

7.3 Rot-Schwarz-Bäume

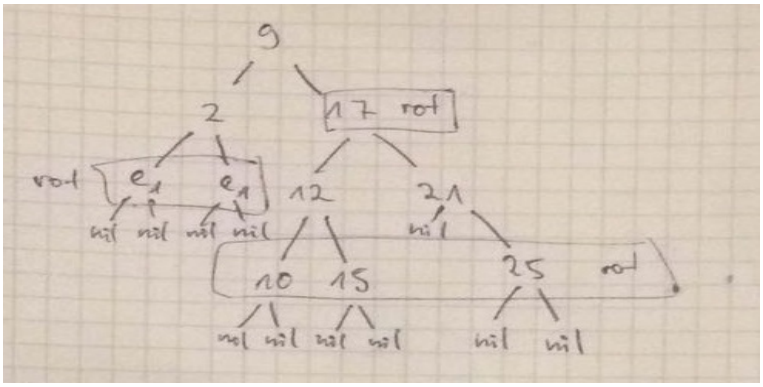
Rot-Schwarz-Bäume sind eine Möglichkeit, um balancierte Suchbäume zu implementieren. Wir betrachten nun binäre Suchbäume, deren Knoten folgende Felder enthalten: *color, key, left, right* und *p* (Parent). Alle Knoten werden als innere Knoten betrachtet, die Blätter durch Zeiger auf `nil` dargestellt. Ein binärer Suchbaum, der folgende Eigenschaften erfüllt, ist ein Rot-Schwarz-Baum:

1. Jeder Knoten ist schwarz oder rot.
2. Die Wurzel ist schwarz.
3. Jedes Blatt (`nil`) ist schwarz.
4. Falls ein Knoten rot ist, so sind beide Kinder schwarz.
5. Für jeden Knoten gilt: Alle Pfade zu den Nachfahren, die Blätter (`nil`) sind, haben die gleiche Anzahl von schwarzen Knoten.

A) Färben Sie den angegebenen Suchbaum so, dass es sich um einen Rot-Schwarz-Baum handelt.



B) In einem Rot-Schwarz-Baum ist das Einfügen von Elementen eine komplizierte Operation, die eventuell Rebalancierungen und Umfärbungen nach sich zieht. Fügen Sie zwei geeignete Elemente so in den Suchbaum ein, dass Ihre Färbung aus Teil (a) erhalten bleibt und trotzdem ein gültiger Rot-Schwarz-Baum entstehen.



C) Zeigen Sie, dass ein Red-Black-Tree höchstens Tiefe $2\log(n+1)$ hat. Dabei ist n die Anzahl der (inneren) Knoten.

Da die Rot-Schwarz Bedingung des Baums zur halben Ausgeglichenheit zwingst wegen E.5(-E.1) (gleiche Anzahl schwarzer Knoten im Baum ∀ Pfade), gilt es bei fast voller Ausgeglichenheit $log(n+1)$ Tiefe (wegen der Wurzel die schwarz sein muss und keinen linken Teilbaum haben könnte.) Da nun aber jeder zweite Teilbaum aber lineare Liste werden kann, kann die doppelte Tiefe, also $2 * log(n+1)$ erreicht werden.

D) Ist jeder Rot-Schwarz-Baum auch ein AVL-Baum? Begründen Sie Ihre Antwort.

Nein Gegenbeispiel siehe geg. Rot-Schwarz-Baum aus a)

7.4 Implementation von Binären Suchbäumen

Laden Sie die Java-Vorlage aus dem Moodle herunter und implementieren Sie:

A) Die Klasse `BSTreeNode`, die das gegebene Interface `TreeNode` implementiert.

B) Die fünf Methoden der Klasse `BSTree`, die keine Implementierung haben.

8 Blatt 07

8.1 Anwendung von AVL-Bäumen

Betrachten Sie folgendes Problem: Gegeben eine Menge C von Kreisen:

$$C = \{c_i = (x_i, y_i, r_i) \mid i = 1, \ldots, n\}.$$

Dabei ist (x_i, y_i) der Mittelpunkt und $r_i > 0$ der Radius des Kreises c_i. Sie können annehmen, dass sich keine drei Kreise in einem Punkt schneiden und dass es keinen Kreis gibt, der vollkommen in einem anderen Kreis enthalten ist. Gefragt ist nun nach dem *Schnittproblem*, das heißt nach allen Schnittpunkten der Kreise in C. Für Ihre Analyse dürfen Sie annehmen, dass die Schnittpunkte von zwei Kreisen mit einer gegebenen Funktion `Schnitt(`c_i`,`c_j`)` in Zeit $O(1)$ berechnet werden können. Geben Sie einen *Plane-Sweep* Algorithmus für das Schnittproblem in Pseudocode an. Halten Sie sich dabei an folgende Hilfestellungen: Verwenden Sie eine X-Struktur und eine Y-Struktur (beides AVL-Bäume), die „wichtige" x- und y-Werte enthalten. Geben Sie die Laufzeit Ihres Algorithmus an und begründen Sie die Korrektheit Ihres Algorithmus.

[H] **Data** C (Menge von Kreisen)
Result gesamte Anzahl an Schnittpunkten der Kreise
Sei A und B ein leerer AVL-Baum
Sei v und w eine leere Liste
Sortiere C nach x-Werten (x-Werte: $x = x_i - r_i$)
for alle Kreise c_i **do**
| Berechne linkesten und rechtesten x-Wert von c_i
| Füge x-Werte nach ihren Werten in A ein
| Berechne obersten und untersten y-Wert von c_i
| Füge y-Werte nach ihren Werten in B ein
end
v ← InterceptionFinder(C,A,B)
for alle Verbindungen von Kreisen in v **do**
| Schnitt(c_i, c_j) schreibe in w
end
return w

InterceptionFinder [H] **Data** C (Menge von Kreisen), A und B (AVL-Bäume)
Result eine Liste R aller Verbindungen der Kreise
for jedes c_i in C **do**
| Suche linkeste und rechteste x-Wert in A
| Suche oberste und unterste y-Wert in B
| überprüfe alle Kanten von den vier Punkten
if Kanten der Punkte in A und B die selben Kreise verbinden
| schreiben Verbindung in Liste v
end

end

n gibt Die Anzahl der Kreise in C an. Der Algorithmus sortiert einmal die Menge der Kreise nach dem am linkesten liegendem x-Wert mit einer Laufzeit von $O(n * log(n))$. Dann durchläuft er die Liste von links nach rechts und bestimmt für jeden Kreis vier Werte: die äußersten liegenden Werte (linkeste, rechteste,oberste, unterste) mit $O(4 * n) \Rightarrow O(n)$. Die Werte werden dann in die AVL-Bäume eingetragen (worst case: $O(log(n))$). Danach wird unser 2.Algorithmus der InterceptionFinder aufgerufen: er findet Kantenverbindungen der Knoten im Baum eines Kreises zu den Knoten der anderen Kreise. Er speichert sie in einer Liste nur, wenn sie in beiden AVL-Bäume vorkommen. D.h. ein Kreis gilt als geschnitten, wenn einer der x-Werte des ersten Kreises mit einem der x-Werte des zweiten Kreises eine gemeinsame Kante besitzen und dasselbe muss auch für einen y-Wert gelten. Also hat man mit einer linearen Suche, weil die Liste unsortiert ist eine Laufzeit von $O(n)$. Das Finden der Kanten läuft hingegen in konstanter Zeit $O(1)$. Um die Liste aller Verbindungen aufzurufen, wird wieder eine konstante Laufzeit für das Problem gelöst. (Worst case: $O(n)$).

Damit beträgt die Laufzeit:
$O(n * log(n)) + O(n) + O(log(n)) + O(n) + O(1) + O(n) = O(n * log(n))$

8.1.1 Korrektur

in X-Structure we save events ordered by x-value

- start of circle
- end of cirlce
- intersections of different circles

in Y-Structure we save semi-circles ordered by y-value

- start of circle divide the circle into two semi-circles c_l and c_u
  insert both into Y-structure
  determine the intersections with predecessor and successor in Y-structure
  insert new intersection-events into X-structure
- end of circle determine intersections of predecessor and successor of current circle
  new intersection-events into X-structure
  delete the current circle in the Y-structure
- intersection of different circles save intersections
  suppose two semi-circles c_i and c_j are intersecting, where c_j is successor of c_i
  determine intersections
  switch c_i and c_j in the Y-structure

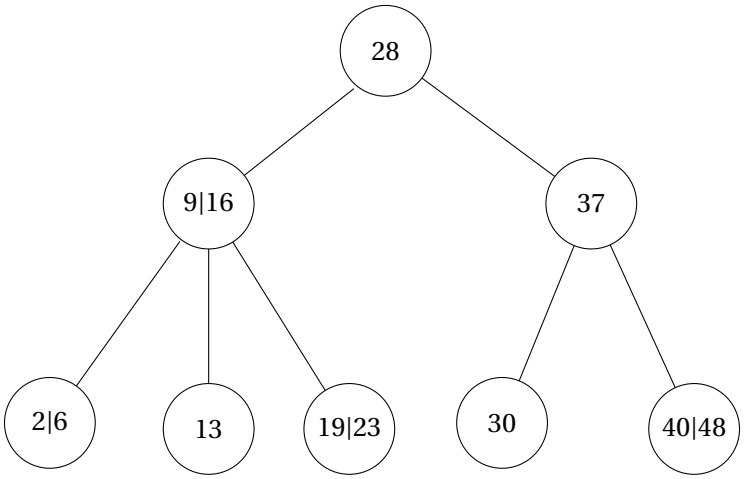
intersections(C) [H] initialize list P
fill X-Structure with start and end points of all circles

X not empty get and remove first event e in X e is start of circle create lower circle c_l and upper circle c_u from current circle insert c_l and c_u into Y-strcture determine intersections of c_l with its predecessor in Y-structure add intersections events to X determine intersections of c_r with ist successor in Y-structure add intersection events to X e is end of circle let c be current circle determine intersections of predecessor c_l with successor c_u insert new intersection evens to X delete current circle from Y-structure e is intersection of different circles add the new intersection to P let c_i and c_j = successor c_i be the corresponding crossing semi-sircles determine intersections of c_i with successor c_j add new intersection events to X switch c_i and c_j in Y-structure **return** P;

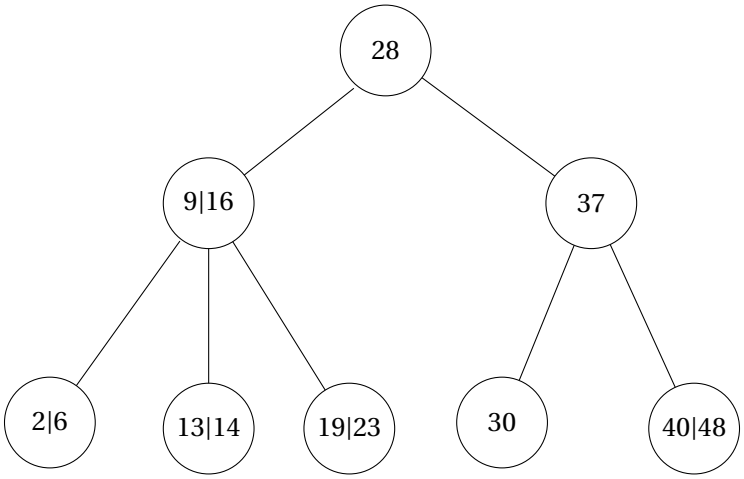
8.2 B-BÄUME

In dieser Aufgabe geht es um B-Bäume der Ordnung 2, d. h. ein innerer Knoten hat mindestens zwei und höchstens drei Kinder. Gegeben ist der folgende B-Baum der Ordnung 2:

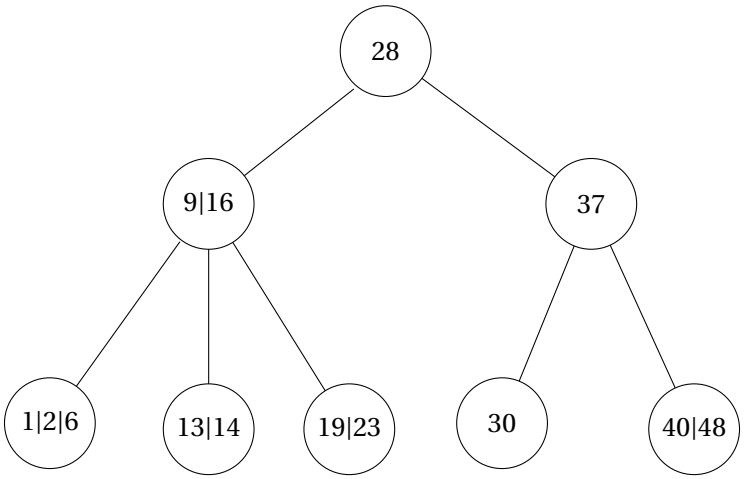
A) Fügen Sie die folgenden Elemente in den gegebenen B-Baum ein: Zuerst 14 und dann 1.



14 einfügen



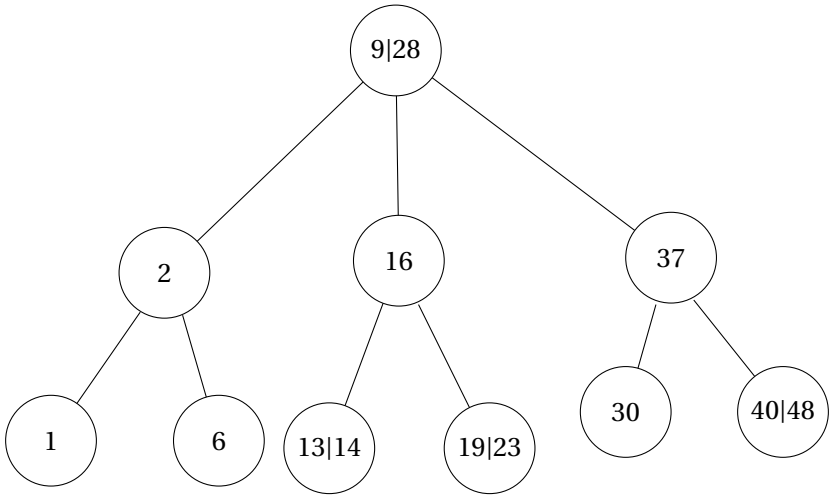
1 einfügen



Knoten (1|2|6) hat zu viele Elemente → Ordnung 2 nicht mehr erfüllt

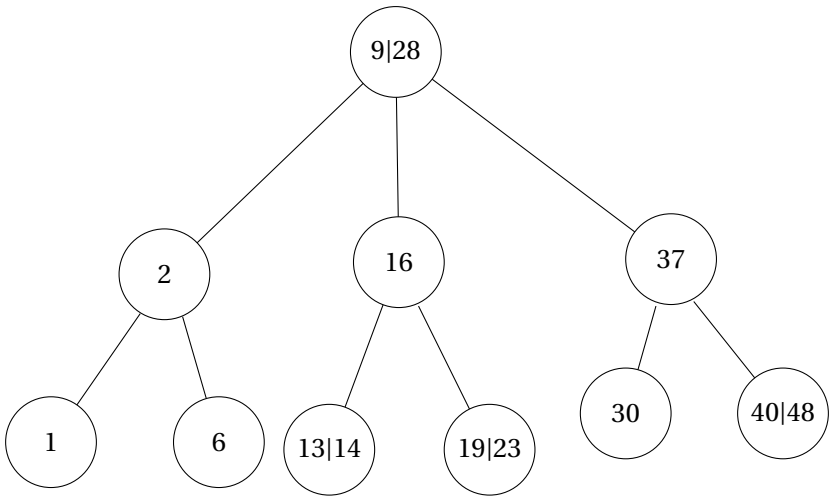
teile den Knoten (1|2|6) und lasse die 2 nach oben steigen

Knoten (2|9|16) hat dann wiederum auch zu viele Elemente
→ aufspalten und 9 nach oben wandern

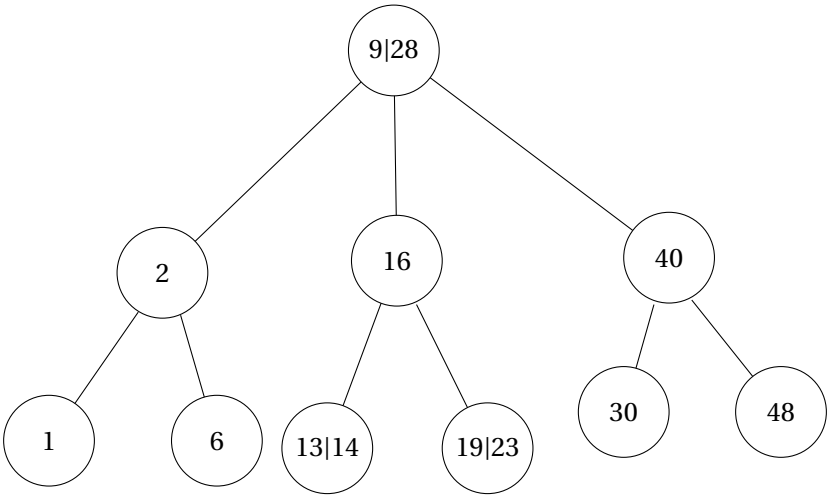


B) Löschen Sie die folgenden Elemente aus dem B-Baum, den Sie in Teil (a) erhalten haben: 37, 40, 19, 23 (in dieser Reihenfolge).

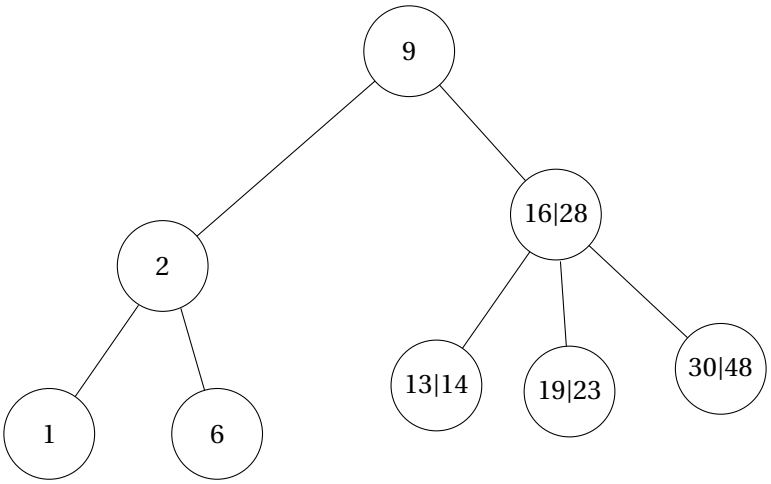
Baum aus a)



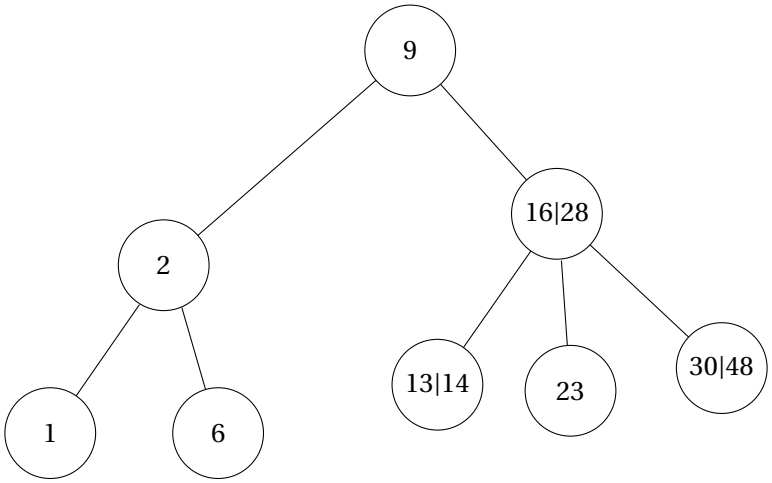
lösche 37 und nehme 40 als Ersatzknoten



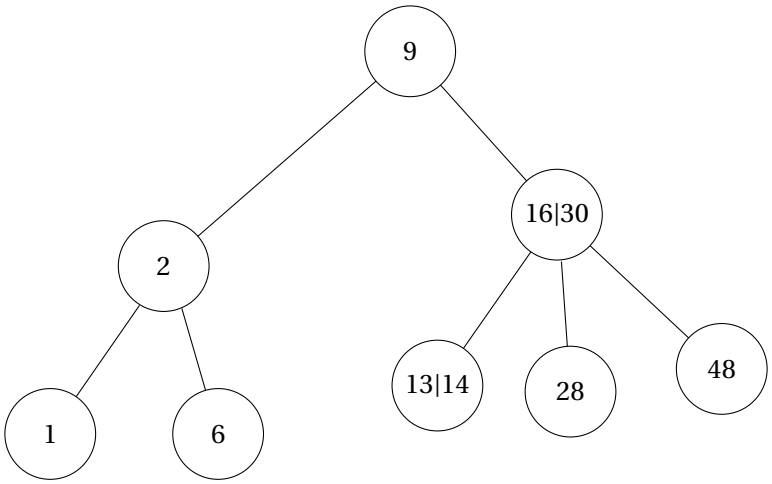
lösche 40: 30 oder 48 kann man nicht als Ersatzknoten wählen, da sonst die B-Baum Bedingungen nicht mehr erfüllt werden, d.h. verschmelze also (30|48) zu einem Knoten und suche nach einem Vaterersatz: 28. Dieser sinkt ein Knoten nach unten und verschmilzt mit dem Knoten 16.



lösche 19



lösche 23: Wenn man den Knoten 23 löscht, dann würde dem Knoten (16|28) ein Kind fehlen, d.h. wir löschen 23 lassen 28 nach unten wandern und 30 nach oben.



Begründen Sie jeweils die Korrektheit und Laufzeit Ihres Algorithmus.

A) Seien S und T zwei (2,4)-Bäume, wobei alle Elemente aus S kleiner sind als die Elemente aus T, d.h. $\max S < \min T$. Geben Sie einen effizienten Pseudocode-Algorithmus `merge`(S,T) an, der einen (2,4)-Baum aus der Vereinigung von S und T erstellt.

merge(S,T)
[H] m ← biggest element in tree S
height(S) ≤ height(T) T' ← T height(S) ≠ height(T') T' ← leftmost child of root(T') merge root(S) m and v= root(T') split tree T starting from v **return** T S' ← S height(S') ≠

height(T) S' ← rightmost child of root(S') merge root(T) m and u=root(S') split the tree S starting from u **return** S

B) Geben Sie einen effizienten Pseudocode-Algorithmus `split(`T, x`)` an, der einen (2,4)-Baum T am Wert x in zwei (2,4)-Bäume $T_{\leq x}$ und $T_{>x}$ aufspaltet. Dabei soll $T_{\leq x}$ alle Elemente aus T enthalten, die kleiner oder gleich x sind und $T_{>x}$ alle Elemente aus T, die größer als x sind.

split(S,x):
[H] l,r ← empty list
w ← root(T) w is not leaf i ← 1 i

8.4 Amortisierte Analyse

Eine *Queue* ist eine Datenstruktur, die die beiden Operationen `enqueue` und `dequeue` unterstützt:

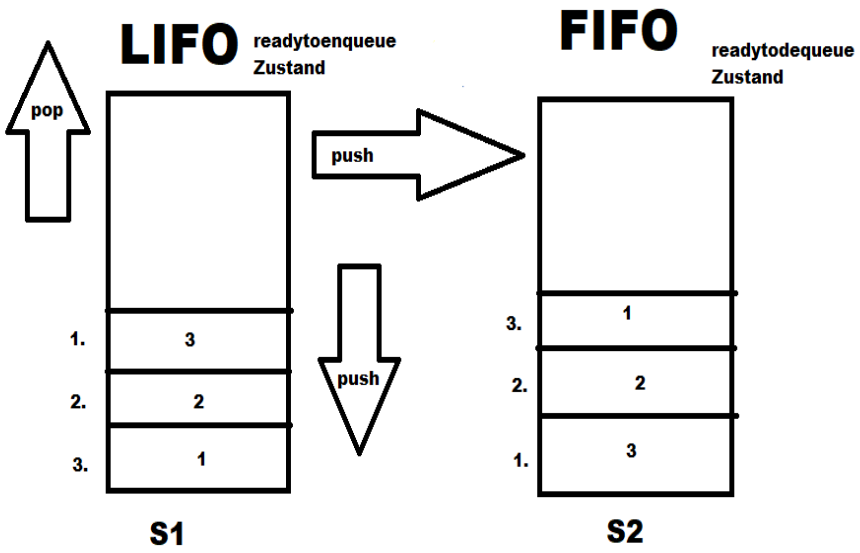
- `enqueue` zum Hinzufügen eines Objekts und
- `dequeue` zum Zurückholen und Entfernen eines Objektes.

Dabei wird nach dem Prinzip *first-in-first-out* gearbeitet, das heißt, es wird von `dequeue` immer das Objekt aus der Queue zurückgegeben, welches von den in der Queue noch vorhandenen Objekten als erstes mit `enqueue` hineingelegt wurde.
Ein *Stack* ist eine Datenstruktur, die die drei Operationen `push`, `pop` und `empty` unterstützt:

- `push` zum Hinzufügen eines Objekts,
- `pop` zum Zurückholen und Entfernen eines Objektes, und
- `empty` zum Prüfen, ob der Stack leer ist.

Dabei wird nach dem Prinzip *last-in-first-out* gearbeitet, das heißt, es wird von `pop` immer das Objekt aus dem Stack zurückgegeben, welches von den in dem Stack noch vorhandenen Objekten als letztes mit `push` hineingelegt wurde.

A) Beschreiben Sie, wie man eine Queue mit Hilfe von zwei Stacks simulieren kann.



Es gibt 2 Zustände für die Stacks:

- readytoenqueue, wenn S2 leer ist
- readytodequeue, wenn S1 leer ist

Hierbei wird in S1 die Reihenfolge für das Hinzufügen festgelegt und S2 muss hierfür leer sein. Soll dequeued werden, so gilt FIFO, also muss S1 und S2 gepusht werden, falls S2 zu Beginn dieser Operation leer war und anschließend stehen in S2 die Elemente aus S1 in verkehrter Reihenfolge, so dass man im readytodequeue Zustand landet und beliebig oft hintereinander FIFO ausgeben kann. Soll nun wieder enqueued werden, muss S2 wieder nach S1 gepushed werden, um in den readytoenqueue Zustand zu gelangen.

enqueue(x):
[H] **if** (not(empty(S1))):
do push(S2,S1)
enqueue(x)
else
do push(x,S1)
dequeue(x):
[H] **if** (not(empty(S1))):
do push(S1,S2)
dequeue(x)

else
do pop(S2)

B) Zeigen Sie, dass die amortisierte Laufzeit für jede `enqueue`- und `dequeue`-Operation von dieser simulierten Queue $O(1)$ ist.

	Einzahlung	Entsumme	Saldo/Ergebnis
readytoqueue	kostet push(S2,S1) und $O(1) * n$	$O(1) * n$	push(S2,S1)
readytodequeue	$O(1) * n$	push(S1,S2) und $O(1) * n$	push(S1,S2)

Da push(s_i, s_j) n-Schritte für n Elemente benötigt ist die Laufzeit $O(1)$
⇒ Saldo/Ergebnis der Account Methode der armortisiertes Laufzeitanalyse liegt in O(1).

9 Tutorium 18.06.2018: Dynamisches Programmieren

Beispiel: Fibonacci
[H] Fib(x)
fibarr[] = new Int[x]
fibarr[0]=0
fibarr[1]=1
for(i=2 to x)
fibarr[i]=fibarr[i-1]+fibarr[i-2]
return fibarr[x]

10 Blatt 08

10.1 Skiplists

Fügen Sie folgenden Elemente in eine Skiplist ein:

$$17, 3, 7, 11, 13, 19, 2, 23, 27.$$

Es stehen ihnen dazu folgende Zufallsbits zur Verfügung (von links nach rechts verbrauchen):

$$1101100101000110001.$$

Erhöhen Sie L_i so lange bis das Zufallsbit 1 ist. Geben Sie Ihre Datenstruktur nach jeder Einfügeoperation an.

Bemerkung: Leider hat es uns zeitlich nicht gereicht, alle Querlinien einzufügen.

leere Skiplist

$-\infty+\infty$

17,ZB:1 verbleibende ZB: 1011 0010 1000 1100 01

$-\infty17+\infty$

3, ZB:1 verbleibende ZB: 0110 0101 0001 1000 1

$-\infty317+\infty$

7, ZB:01 → Levelerhöhung um 1; ZB: 1001 0100 0110 001

$-\infty-\infty3717+\infty+\infty7$

11, ZB:1 v. ZB: 0010 1000 1100 01

$-\infty-\infty371117+\infty+\infty7$

13, ZB: 001 → Levelerhöhung um 1, v. ZB: 0100 0110 001

$-\infty-\infty-\infty37111317+\infty+\infty+\infty71313$

19, ZB:01 v ZB: 0001 1000 1

$-\infty-\infty-\infty3711131719+\infty+\infty+\infty7131319$

2, ZB: 0001 Levelerhöhung um 1; v. ZB: 10001

$-\infty-\infty-\infty-\infty23711131719+\infty+\infty+\infty+\infty7131319222$

23, ZB:1 v. ZB: 0001

$-\infty-\infty-\infty-\infty2371113171923+\infty+\infty+\infty+\infty7131319222$

27, ZB 0001 v.ZB: empty

$-\infty-\infty-\infty-\infty237111317192327+\infty+\infty+\infty+\infty7131319222272727$

10.2 Skiplists und Binäre Suchbäume

Sei T ein binärer Suchbaum der Größe n, der nicht notwendigerweise balanciert ist. Beschreiben Sie einen Algorithmus in Pseudocode, der in Zeit (n) aus T eine Skipliste S bildet. Dabei soll S so aufgebaut sein, dass die Suche in S eine deterministische worst-case Laufzeit von $(\log n)$ hat. Begründen Sie die Korrektheit Ihres Algorithmus.

Für $O(logn)$ Laufzeit in Skiplist wird folgende Levelverteilung der Elemente benötigt:

- das Mitteelement muss das höchste Level n erreichen
- die erhaltenen zwei Skiplisten am Mitteelement erreichen das nächsthöchste Level n-1
- mit dem nächsten 4 erhaltenen Skiplisten wird auf diese Weise verfahren, bis die Skiplisten der Länge 1 schließlich als unterstes Level 1 erreichen

Durch die unterschiedlichen Levelhöhen der immer mittleren Elemente wird durch Abwärtsbewegung von oberstes Level oder Vorwärtsbewegung effizient indirekt immer $\frac{n}{2}+1$ der Skipliste verworfen, so dass die Suche in solch einer Skiplistenverteilung genau der binären Suche entspricht, welche $O(logn)$ Laufzeit hat.
[H] // function to-skiplist(T) S ← sort-to-list(T)
level-up(S)

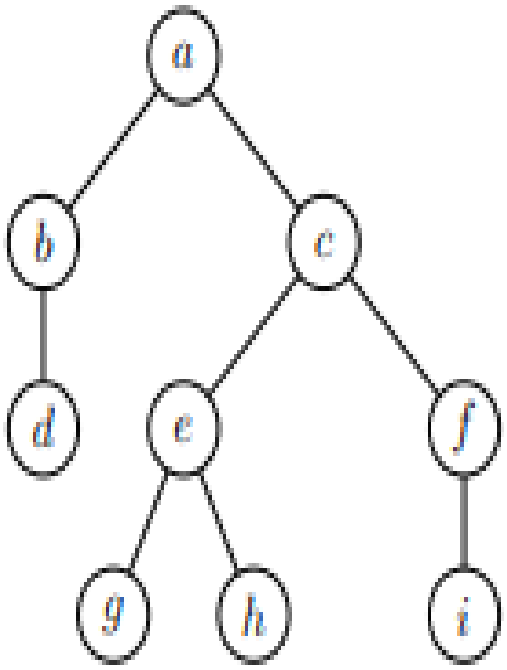
// function sort-to-list(T)
sorted-list ← list(sort-to-list(left child),root,sort-to-list(right child)) sorted-list
// function level-up(S)
i=0, i++, $i < log(i)$ raise-level-of: median(S) ← log(i)-i continue-with-new-lists-S_i S

In to-skiplist(T) wird sort-to-list aufgerufen, welches den binären Suchbaum T rekursiv in ein (flache) Skipliste S in n Schritten einfügt und das Ergebnis an level-up(S) weitergibtm welches in (log n)-Schritten die Skiplisten Level von S anpasst. Wir haben also eine Laufzeit von $O(n)+O(logn) \in O(n)$.

10.3 Dynamisches Programmieren

Gegeben ein Baum $T = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten zwischen diesen Knoten sind. Eine *unabhängige Menge* ist eine Menge $M \subseteq V$, sodass für alle $v \in M$ gilt, dass $w \notin M$, falls es eine Kante $\{v, w\} \in E$ gibt. Die unabhängige Menge M ist maximal, falls es keine andere unabhängige Menge gibt, die mehr Knoten als M enthält.

A) Geben Sie eine maximale unabhängige Menge für das folgende Beispiel an:



Beispiel für eine maximal unabhängige Menge: M=c,d,g,h,i

B) Beschreiben Sie einen Algorithmus mit Dynamischem Programmieren, der eine maximale unabhängige Menge von T berechnet.

First we develop an algorithm to calculate the number of nodes in a maximum independent set. To this end we use variables $v.max$, which will contain the number of nodes of a maximum independent set of the subtree rooted at a node $v \in V$. For our algorithm we observe the following. For a node $v \in V$, there are two possibilities:

1. Either v is int the independent set. In this case no children of v can be in the independent set, but the children of the children of v can. Thus, if the grandchildren of v are $w_1 ... w_k$ we

conclude

$$v.max = 1 + \sum_{i=1}^{k} w_i.max$$

in this case assuming that we already know the values $w_i.max$ for all $i = 1...k$.

2. Otherwise v is not contained in the independent set. In this case the children of v might be in the independent set. Let $c_1...c_l$ be the children of v. Then we have

$$v.max = \sum_{i=1}^{l} c_i.max.$$

To calculate the maximum set of v, we have to use the possibility, where we can obtain the higher value for v.max:

$$c.max = max \sum_{i=1}^{l} c_i.max.1 + \sum_{i=1}^{k} w_k.max$$

SizeMaxIndependentSet(T)
[H] // initizalize v.max to 0 for all nodes in the tree T:
not all nodes of T are visited in a post-order walk e has no children v.max $\leftarrow 1$ v has no grandchildren v.max $\leftarrow \sum c.max$ v.max $\leftarrow max \sum c.max.1 + \sum w.max$

MaxIndependentSet(T)
[H] // initialize an empty set M
// initialize an empty list l and insert the root of T into it:
l is not empty v $\leftarrow$ first element of l delete the first element of l r has no children M $\leftarrow M \cup r$ r has no grandchildren M $\leftarrow M \cup c, cischildofr$ r.max = $\sum c.max$ append all children of r to l M $\leftarrow M \cup r$ append all grandchildren of r to l M

C) Begründen Sie die Korrektheit Ihres Algorithmus.

D) Wie ist die Laufzeit Ihres Verfahrens?

10.4 Dynamisches Prog.: Palindrom

Ein *Palindrom* ist eine Zeichenfolge, die vorwärts und rückwärts gelesen gleich ist. Beispiele für Palindrome sind alle Zeichenfolgen der Länge 1, `KAJAK`, `REITTIER`, `SEI FIES` und `DREH MAL AM HERD`.

Gegeben sei eine Zeichenfolge $S[1..n]$. Eine *Teilfolge* S' von S besteht aus Buchstaben, die in S' in der selben Reihenfolge auftreten wie in S. Zum Beispiel ist `ATMEN` eine Teilfolge von

ALGORITHMEN. Das *längste Palindrom* in S ist die längste Teilfolge von S, die ein Palindrom darstellt.

A) Geben Sie einen Algorithmus in Pseudocode an, der die Länge des längste Palindrom in S berechnet. Verwenden Sie dabei Dynamische Programmierung.

Länge des längsten Palindrom
[H] String S int Länge des längsten Palindroms in S // löschen aller Leerzeichen
I=1 ... length(S) i-tes Zeichen von S ist Leerzeichen S ← konkateniere die ersten i-1 Zeichen von S mit den letzten length(S)-i Zeichen int i ← 1
int j ← length(S)
LLPWorker(S,i,j)

LLPWorker
[H] String S, int i, int j int Länge des längsten Palindroms in S // nur ein Zeichen → palindromisch
i == y 1
// nur 2 gleiche Zeichen → palindromisch
length(S) == 2 und S[i] == S[j] 2
// erstes und letztes Zeichen gleich
S[i] == S[j] LLPWorker(S, i+1,j-1) +2
// erstes und letztes Zeichen verschieden
max(LLPWorker(S,i,j-1),LLPWorker(S,i+1,j))

max
[H] int x, int y int (max von a und b) a > b a b

B) Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus aus (a).

Korrektheit
Zuerst müssen alle Leerzeichen aus dem Ausgabestring gelöscht werden. Für die beiden trivialen Fälle liefert unser Algorithmus das richtige Ergebnis, da alle Wörter der Länge 1 palindromisch sind und ein Wort der Länge 2 ebenfalls palindromisch ist, falls die Zeichen übereinstimmen. Wenn die Zeichen aber verschieden sind und wir noch ein Wort der Länge 2 haben so landen wir im 4.Fall, so wird das Maximum aus dem längsten Palindrom zweier einbuchstabiger Wörter gebildet.
Um den dritten Fall zu erklären, wo das erste und letzte Zeichen identisch sind, müssen wir LLPWorker, auf den String zwischen dem ersten und letzten Zeichen, aufrufen und 2 addieren.
Beim letzten Fall, sind das erste und das letzte Zeichen verschieden, d.h. sie verlängern nicht die Länge des Polyndroms. Also überprüfen wir das Maximum aus den LLPWorker von dem String aus dem ersten Zeichen bis zum vorletzten Zeichen und dem LLPWorker vom String

vom 2. Zeichen bis zum letzten.

Laufzeit

- Leerzeichen löschen mit Laufzeit $O(n)$
- LLPWorker
  - 1.Fall T(1)=1
  - 2.Fall T(2)=1
  - 3.Fall T(n-2)
  - 4.Fall T(n-1)

  Um also die Laufzeit für den worst case abschätzen zu können, brauchen wir eigentlich nur den 4. Fall anschauen, so das der String sozusagen immer weiter in den 4.Fall kommt und die Zeichen nicht identisch werden.

  Daraus folgt das Rekursionsschema:

$$T(n) = 2T(n-1)$$
$$= 2 * 2T(n-2)$$
$$= 2 * 2 * 2T(n-3)$$
$$= 2^i * T(n-i)$$

  Jetzt brauchen wir das Rekursionsende:

$$T(n)_{n-2} = 2^{n-2}T(n-(n-2)) = 2^{n-2}T(n-n+2) = 2^{n-2}T(2) = 2^{n-2}$$

  D.h. wir gehen von einer Laufzeit von $O(2^{n-2}) \in O(2^n)$ aus.

C) Geben Sie einen Algorithmus in Pseudocode an, der mit Hilfe von (a) das längste Palindrom in S berechnet und begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.

D) Demonstrieren Sie die Funktionsweise Ihrer Algorithmen an der Zeichenfolge `DADBBCAA`.

11 BLATT 09

11.1 DYNAMISCHES PROGRAMMIEREN

Seien $u, v, w \in \Sigma^*$ Wörter über einem Alphabet Σ. Wir nennen *w einen Shuffle* aus u und v, wenn es eine Zerlegung $u = u_1 \dots u_k$ und $v = v_1 \dots v_l$ gibt mit $u_i, v_j \in \Sigma^*$ für alle i, j, so dass $w = u_1 v_1 u_2 v_2 \dots$ ist.
Zum Beispiel ist w = *INaFOlgoRMrithAmTIenK* ein Shuffle aus den Wörtern u = *INFORMATIK* und v = *algorithmen*.

A) Geben Sie einen Algorithmus an, der für drei gegebene Wörter u, v, w unter der Verwendung von dynamischer Programmierung entscheidet, ob w ein Shuffle aus u und v ist.

Algorithmus testet, ob $w \backslash u = v$, also die Symbole von u werden abgeleitet von links nach rechts, dabei werden all diese Symbole aus w entfernt. Nach Abarbeitung von u, muss w=v gelten, falls w Shuffle war. u,v,w werden als Array repräsentiert.

checkShuffle(w,u,v)
[H] i=0; $i < length(u)$; i++ delete($u[i], w$) $w00v$ true false

delete($u[i], w$)
[H] j = 0
$j < length(w)$ $u[i] == w[j]$ k=j;$k < length(w)-1$; k++ $w[j] = w[j+1]$ checkShuffle(w,u,v) j++ false

B) Geben Sie die Laufzeit in Abhängigkeit der Längen der Wörter an.

Beim Algorithmus wir u einmal komplett durchlaufen ($O(lenght(u))$), wobei zwischen jedem Schritt, w entweder komplett durchlaufen wird und Ende oder in w das Symbol aus u gefunden in maximal length(w) Schritten.
Da nun length(w) bei gefundenen Symbol von u in w nicht um 1 verringert, wird length(w) jeden Schritt bei der Abarbeitung von u, wird im nächsten Schritt immer nur die Teilwertlänge von v weitergelaufen, welche zwischen 1 und lenght(v) liegt. Im worst case haben wir also $O(length(u)) + O(length(v))$ und die $O(length(w))$ entspricht mit length(w)=n erhalten wird $O(n)$ Laufzeit (mit Faktor $c \leq 1$).

11.2 PROGRAMMIERAUFGABE

Laden Sie die Java-Vorlage aus dem Moodle herunter und implementieren Sie die folgenden Methoden:

A) `isShuffled(String w, String u, String v)`, die entscheidet ob das Wort w ein Shuffle aus den zwei Wörtern u und v ist.

B) `isShuffled(String w, String u, String v, String z)`, die entscheidet of das Wort w ein Shuffle aus den drei Wörtern u, v und z ist.

11.3 Anagramme

Seien u und v zwei Wörter über einem endlichen Alphabet Σ. Ein Anagramm eines Wortes u ist ein Wort v, das aus einer Permutation der Buchstaben von u gebildet werden kann.

A) Welche Eigenschaften muss eine Hashfunktion erfüllen, um Anagramme schnell identifizieren zu können?

Für jedes Symbol/ Buchstabe aus Wort muss ein individueller Wert existieren (Zahlenwert). Zusätzlich darf/ sollte keiner dieser Werte aus der Summe anderer Werte entstehen (Wertunabhängigkeit).

B) Die Funktion `getBytes` weißt jedem Buchstaben aus Σ eine eindeutige natürliche Zahl zu. Betrachte folgende Hashfunktion $h\colon \Sigma^* \to \mathbb{N}$

$$h(w_1 \dots w_n) = \sum_{i=1}^{n} \texttt{getBytes}(w_i)$$

Zeigen Sie, dass die Hashfunktion h zwei Anagramme auf denselben Hashwert abbildet.

Da $h(w_1, ..., w_n) = \sum_{i=1}^{n} getBytes(w_i)$ gilt und $\sum_{i=1}^{n} getBytes(w_i) = w_1 + ... + w_n$, so ist nach Kommutativgesetz die Reihenfolge der Summanden für den Ergebniswert irrelevant, so das alle Permutationen derselbe Summand zum selben Wert führen.

⇒ Zwei Argumente bilden auf denselben Wert ab, da ihre Summanden gleich sind.

C) Warum ist die Funktion dennoch ungeeignet, um Anagramme zu filtern? Welche Strings werden noch auf dieselben Hashwerte abgebildet?

Da zwei Hashwerte summiert den Hashwert eines anderen Symbols erzeugen können, kann z.B. $h(w_1, w_2, w_3) = h(w_1, w_4)$ gelten, falls $w_2 + w_3 = w_4$. Unterschiedliche Werte können also denselben Hashwert erzeugen.

D) Modifizieren Sie die Hashfunktion so, dass sie sich besser dazu eignet Anagramme zu filtern. Begründen Sie, warum sich Ihre neue Hashfunktion besser verhalten sollte und zeigen Sie, dass sie immer noch zwei Anagramme auf denselben Wert abbildet.

Wähle als Hashwert für einzelne Symbole Primzahlen.
$getBytes(w_i)$ soll also für alle existierende Symbole eine Primzahl sein. Da Summen von

Primzahlen immer eindeutig sind. (Beweis: Primfaktorzerlegung, Multiplikationen sind vor Schreibweise für Addition)

11.4 HASHING

Betrachten Sie die folgende Hashfunktion

$$\begin{aligned} h_{a,b,m}: \quad &\to \{0,\ldots,m-1\} \\ &x \mapsto (ax+b) \bmod m \end{aligned}$$

und die Eingabewerte

$$S = \{13, 45, 64, 78, 116\} \ .$$

A) Geben Sie Werte für a, b und m an, sodass

I.) alle Werte aus S auf den selben Hashwert abgebildet werden.

- a=0
- b=4
- m=3

Dann ist $h_{0,4,3}(x)$ eine Konstante, die alle Werte auf 4 mod 3 = 1 abbildet.

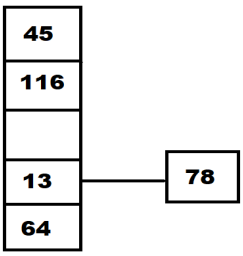
II.) alle Werte aus S auf unterschiedliche Hashwerte abgebildet werden.

- a=1
- b=0
- m=10

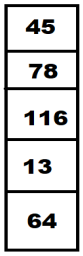
Dann ist $h_{1,0,10}(13) = 3$, $h_{1,0,10}(45) = 5$, $h_{1,0,10}(64) = 4$, $h_{1,0,10}(78) = 8$ und $h_{1,0,10}(116) = 6$, also alles verschiedene Werte.

B) Sei $g_i(x) := (h_{1,0,5}(x) + i) \bmod 5$. Geben Sie jeweils die Hashtabelle an, die entsteht, wenn S in aufsteigender Reihenfolge in eine zuvor leere Hashtabelle eingefügt wird. Verwenden Sie dabei die folgenden Kollisionsauflösungsstrategien:

I.) Verkettetes Hashing mit der Hashfunktion g_0



II.) Hashing mit offener Adressierung mit $(g_i)_{i\geq 0}$



C) Zeigen Sie, dass es für alle Werte von a und b bei $m = 5$ zu Kollisionen kommt.

Beweis:
Seien a und b beliebig
Umformen der mod-Rechenregeln:
$h_{a,b,5}(x) = (a * x + b) mod 5 = (((a * x) mod 5) + b) mod 5$
$= (((x * a) mod 5) + b) mod 5 = ((((x mod 5) * a) mod 5) + b) mod 5$
Einmaliges Einsetzen für x die Werte 13 und 78
zz. das folgende Gleichung für alle a,b gilt :
$((((13 mod 5) * a) mod 5) + b) mod 5 = ((((78 mod 5) * a) mod 5) + b) mod 5$
$\Leftrightarrow (((3 * a) mod 5) + b) mod 5 = ((((3 * a) mod 5) + b) mod 5$

Es gibt also unendlich viele Lösungen für diese Gleichung, d.h. diese gilt für alle a,b. Daraus folgt, dass es immer zur Kollision bei $m = 5$ kommt.

D) Geben Sie Werte für a, b und m an, sodass m minimal ist und alle Werte aus S auf unterschiedliche Hashwerte abgebildet werden.

- a=1
- b=0
- m=6

Kontrolle:

- 13 mod 6 =1
- 45 mod 6 = 3
- 64 mod 6 = 4
- 78 mod 6 = 0
- 116 mod 6 = 2

Man sieht, dass $m = 6$ minimal ist, da wir bei $m = 5$ in c) gesehen haben, dass es da nicht möglich war und das es bei kleineren m's auch nicht möglich wäre, da man dann auf zu wenigen unterschiedlichen Fächern abbilden könnte.

12 Blatt 10

12.1 Grapheigenschaften

Gegeben sei ein einfacher, zusammenhängender Graph $G = (V, E)$ mit $|V| = n$ und $|E| = m$.

G=(V,E) einfach zusammenhängend
$|V| = n$ Knoten
$|E| = m$ Kanten

A) Zeigen Sie, dass es in G zwei Knoten mit demselben Grad gibt.

G hat zwei Knoten vom selben Grad]] $|V| = n$, maximaler Grad p Knoten: n-1; minimaler Grad: 1
→ Schubladenprinzip : $\frac{n}{n-1} Fächer \rightarrow Doppelbelegung$

B) Zeigen Sie, dass G eine gerade Anzahl an Knoten mit ungeradem Grad hat.

G hat eine gerade Anzahl von Knoten mit ungeradem Grad
$\sum_{v}^{v \in V} deg(v) = 2 * |E| \Rightarrow \sum_{v,even} deg(v) + \sum_{v,odd} deg(v) = 2 * |E| \Leftrightarrow \sum_{v,odd} deg(v) = 2 * |E| * \sum_{v,even} deg(v)$

C) G heiß *vollständig*, wenn jeder Knoten durch Kanten mit jedem anderen Knoten verbunden ist. Berechnen Sie die Anzahl der Kanten in einem vollständigen Graphen.

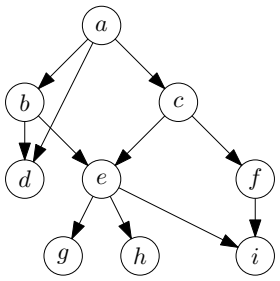
Jeder Knoten hat n-1 Verbindungen $n * n - 1$
ABER jede Kante ist mit zwei Knoten verbunden deshalb gilt: $\frac{n * n - 1}{2}$

D) G heißt *d-regulär*, wenn jeder Knoten aus V einen Grad von d hat. Zeigen Sie, dass $m = \frac{d \cdot n}{2}$, falls G d-regulär ist.

d regulär $\Rightarrow$ deg(v) = d $\forall v \in V$
$\sum_{v} deg(v) = 2 * |E|$
$n * d = 2 * |E|$
$\frac{n * d}{2} = |E| = m$

12.2 Adjazenzmatrix vs. Adjazenzliste

Sei $G = (V, E)$ der folgende gerichtete Graph:

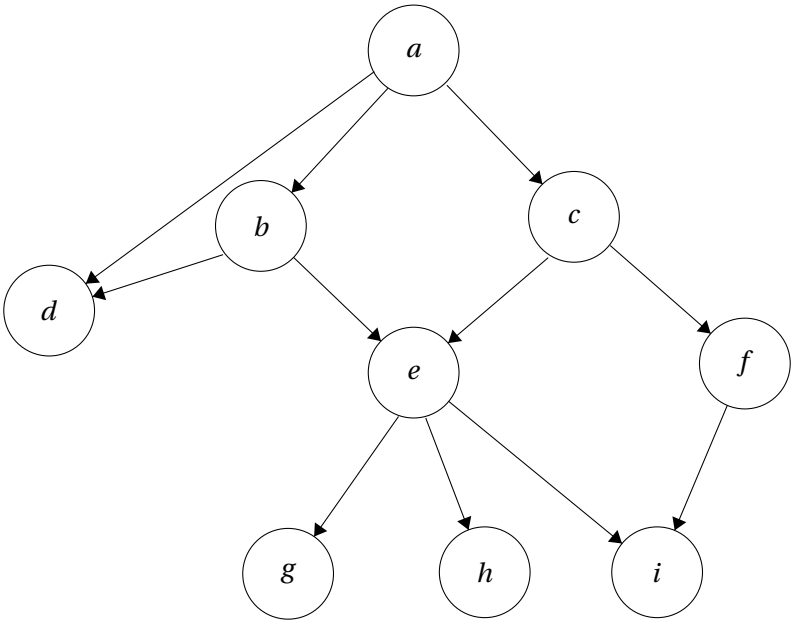


A) Berechnen Sie die Adjazenzmatrix und die Adjazenzliste für G.

Adjazenzliste	Adjazenzmatrix
a → b → c → d	0111 0000 0
b → d → e	0001 1000 0
c → e → f	0000 1100 0
d	0000 0000 0
e → g → h → i	0000 0011 1
f → i	0000 0000 1
h	0000 0000 0
i	0000 0000 0

B) Wenden Sie den Algorithmus aus der Vorlesung auf Ihre Adjazenzliste aus (a) an, um eine topologische Sortierung von G zu berechnen.

Reihenfolge: abcdefghi
Laufzeit: $O(n^2)$



C) Der *transponierte Graph* eines Graphen $G = (V, E)$ ist der Graph $G' = (V, E')$, wobei

$$E' := \{(v, u) \mid (u, v) \in E\}.$$

Nehmen Sie an, dass G als Adjazenzmatrix vorliegt. Beschreiben Sie einen Algorithmus, der den transponierten Graph G' in einer Laufzeit von $(|V|^2)$ berechnet.

transponieren der Matrix mit 2 For-Schleifen

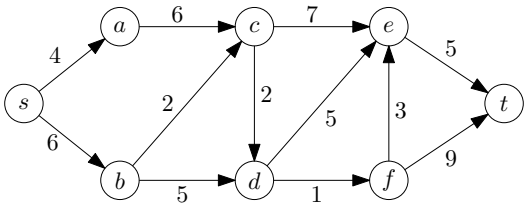
D) Nehmen Sie nun an, dass G als Adjazenzliste vorliegt. Beschreiben Sie einen Algorithmus, der den transponierten Graph G' in einer Laufzeit von $(|V| + |E|)$ berechnet.

Sei $n = |V|$
Sei L Adjeszenzliste für Knoten
i: $L[i]$ enthält Adjeszenztenknoten von i

[H] initialisiere L' für alle Knoten
i= 1 to n j= 1 to $|L[i]|$ $L[i]$ enthält j $L'[j]$.append i

12.3 Dijkstra-Algorithmus

Wenden Sie den Dijkstra-Algorithmus auf das folgende Beispiel an. Verwenden Sie dabei s als Startknoten und geben Sie alle Zwischenschritte an!



	s	s'
1	(s,0,-)	(a,4,s),(b,6,s)
2	(a,4,s)	(b,6,s),(i,10,a)
3	(b,6,s)	(c,8,b),(d,11,b)
4	(c,8,b)	(d,10,c),(e,15,c)
5	(d,10,c)	(f,11,d),(e,15,c)
6	(f,11,d)	(e,14,f),(t,20,f)
7	(e,14,f)	(t,19,e)

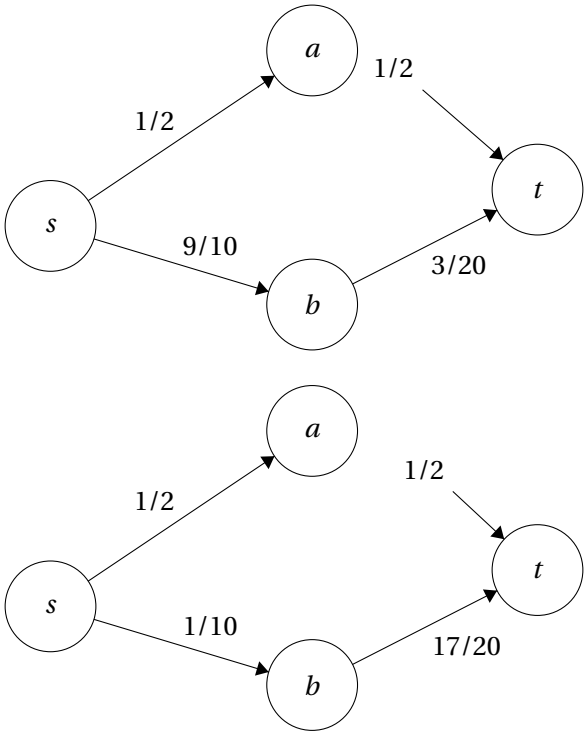
12.4 Anwendung von Dijkstra

Betrachten Sie ein gerichtetes Netzwerk $G = (V, E)$ zusammen mit einer Wahrscheinlichkeitsfunktion $p : E \to (0, 1]$. Dabei ist $0 < p(v, w) \leq 1$ die *Zuverlässigkeit* der Kante (v, w). Seien $s, t \in V$ Knoten, zwischen denen ein Packet gesendet werden soll. Ob das Packet erfolgreich gesendet wird, hängt von der Zuverlässigkeit des verwendeten Pfades ab. Dabei beschreibt die *Zuverlässigkeit eines Pfades* die Wahrscheinlichkeit, dass ein Packet erfolgreich über einen Pfad gesendet werden kann.

A) Wie lässt sich die Zuverlässigkeit eines Pfades $s = v_0 \to v_1 \to \cdots \to v_k = t$ berechnen?

$$\prod_{i=1}^{k} p(v_{i-1}, v_i)$$

B) Geben Sie ein Beispiel an, bei dem der Dijkstra-Algorithmus nicht den Pfad von s nach t mit der höchsten Zuverlässigkeit findet, wenn man für jede Kante (v, w) das Kantengewicht $1 - p(v, w)$ (d. h. die Ausfallswahrscheinlichkeit) verwendet.



- Pfad s-a-t: Kosten 1
- Pfad s-b-t: Kosten $\frac{19}{20}$

C) Beschreiben Sie einen Algorithmus, der mit Hilfe des Dijkstra-Algorithmus, den Pfad zwischen s und t mit der höchsten Zuverlässigkeit findet. Begründen Sie die Laufzeit und Korrektheit Ihres Algorithmus.

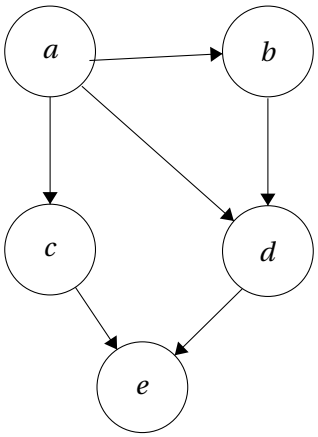
1. Wie bekommen wir Add?
   $log(a * b) = log(a) + log(b)$
2. Wie minimieren?
   Gewicht: $-log(p(v,w))$

13 Tutorium 09.07.2018: Hilfestellung Blatt 11

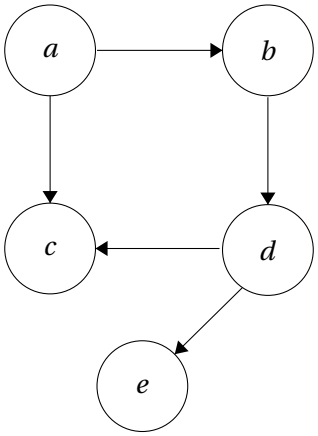
13.1 s-t-Graph

- hat genau eine Quelle (nur ausgehende Kanten)
- hat genau eine Senke (nur eingehende Kanten)

Beispiele:



- a Quelle
- e Senke
- somit s-t-Graph



- a Quelle
- c,e Senke
- somit kein s-t-Graph

13.2 Bellman Ford

[H] für jedes $v \in V$: d(v) = ∞, $Vorgänger$ = empty
d(s) = 0
i = 1 to n-1 $\forall (u,v) \in E$:
d(u)+w(u,v) < d(v) d(v)=d(u)+w(u,v) Vorgänger(v)=u $\forall (u,v) \in E$:
d(u)+w(u,v) < d(v) break new Zykel d vom Endknoten

13.3 Wiederholung Rekursion

Beispiel: Türme von Hanoi

- T(0)=0
- T(1)=1
- T(n)=2(T(n-1))+1

n	Rechnung	T(n)
2	2*1+1	3
3	2*3+1	7
4	2*7+1	15
5	2*15+1	31
...		

$T(n) = 2(2(T(n-2)+1)+1 = 2^2 T(n-2)+2+1$
$= 2^2(2T(n-3)+1)+2+1$
$= 2^3 T(n-3)+4+2+1$
$n-mal = 2^n T(0) + \sum_{i=0}^{n-1} 2^i = 2^n - 1$

Vollständige Induktion:

IA:

- $T(0) = 0 = 2^0 - 1$
- $T(1) = 1 = 2^1 - 1$

IV: Behauptung gelte für beliebig feste $n \in \mathbb{N}$

IS: $n \to n+1$
$T(n+1) = 2(2T(n)+1)+1$
$I.V. = 2(2^n - 1)+1$
$= 2 * 2^n - 2 + 1$
$= 2^{n+1} - 1$

14 Blatt 11

14.1 Algorithmenentwurf

Sei $G = (V, E)$ ein gerichteter Graph. Der Graph G heisst (s, t)-*Graph*, falls er nur eine *Quelle* und nur eine *Senke* hat, wobei ein Knoten Quelle (Senke) heisst, falls er keine eingehenden Kanten hat (nur eingehende Kanten hat).

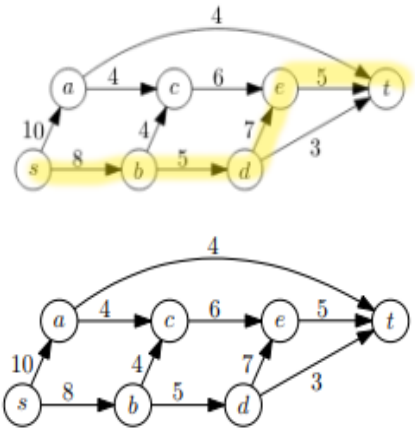
Nehmen Sie an, dass der Graph G in Adjazenzlistendarstellung gegeben ist. Geben Sie einen Algorithmus in Pseudocode, der in Zeit $(|V| + |E|)$ feststellt, ob G ein (s, t)-Graph ist.

[H] every node in G incomingDegree(node) = 0 sinks = 0;
every node in G every target node in nod.adjacentlist comingDegree(targetNode) = incomingDegree(targetNode) + 1 node.adjacentNodes is empty sinks++ sources = 0;
every node in G incomingDegree(node) == 0 sources++ Sources == 1 sinks ==1 true false;

14.2 Maximum Bottleneck Path

Sei $G = (V, E, c)$ ein gerichteter Graph mit positiven Kantenkosten c. Sei P ein Pfad zwischen zwei Knoten u und v, wobei die billigste Kante in P Kosten k habe. P heißt *breitester Pfad* von u nach v, wenn es keinen anderen Pfad in G von u nach v gibt, dessen billigste Kante teurer als k ist.

A) Zeichnen Sie in folgendem Graphen den breitesten Pfad von s nach t ein.



B) Modifizieren Sie Dijkstras Algorithmus so, dass er zu einem gegebenen Startknoten s die breitesten Pfade zu allen anderen Knoten findet. Begründen Sie die Korrektheit und die

Laufzeit Ihres Algorithmus.

[H] visited = s
queue = empty maxHeap sorted by broadestWidth
Forall $(s, v) \in E$ **do**
broadestPathTo(v) = $\{(s, v)\}$ broadestWidth(v) = c(s,v) queue.add(v) **end**
Forall $v \in V | (s, v) \notin E$ **do**
broadestPathTo(v) = {} broadestWidth(v) = 0 **end**
visited != V currentNode = queue.removeMax() visited = visited ∪ currentNode **forall** $(currentNode, next) \in E$ **do**
$next \notin queue$ queue.add(next) minNewWidth = min(broadestWidth(currentNode), c(currentNode, next)) broadestPathTo(next) == {} broadestPathTo(next) = broadestPathTo(currentNode) ∪ (currentNode,next) broadestWidth(next) = minNewWidth minNewWidth > broadestWith(next) broadestPathTo(next) = broadestPathTo(currentNode) ∪ (currentNode,next) broadestWidth(next) = minNewWidth **end**
forall $v \in V$ **do**
(v,broadestWidth(v), broadestPathTo(v)) **end**

Korrektheit:
Unser Algorithmus geht wie der Dijkstra-Algorithmus vor. Es liefert das korrekte Ergebnis, da sich ein Pfad durch das Hinzufügen von Kanten nur verschmälern, aber nicht verbreitern kann. D.h. der lokal breiteste Pfad ist auch der global breiteste Pfad.

Laufzeit:

- while-Schleife: $O(|V|)$
- forall-Schleife: $O(|E|)$
- queue enthält maximal $|V|$ Knoten, d.h. einfügen, sortieren und max entfernen kostet jeweils $O(log|V|)$
- einfügen von $|V|$ Knoten in queue, entfernen von $|V|$ Knoten, updaten des Sortierkriterium um $O(|E|)$
- Laufzeit beträgt: $O(|V|) + O(|E|) + O(2 * |V| + |E|) * O(log(|V|)) = O((|V| + |E|)log(|V|))$

14.3 Bellman/Ford-Algorithmus

Gegeben sei folgender gerichteter Graph $G = (V, E)$. Lösen Sie mit Hilfe des Bellman-Ford-Algorithmus das *single-source shortest-paths* Problem für den Source-Knoten s. Verwenden Sie dabei die folgende Reihenfolge für die Kanten:

$$(s, a), (s, c), (s, d), (a, b), (a, d), (b, a), (c, d), (d, b), (d, e), (e, b).$$

Zeigen Sie alle Zwischenschritte!

Durchlauf 1:

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(v)	Vorgänger(v)
(s,a)	true	1	s
(s,c)	true	2	s
(s,d)	true	2	s
(a,b)	true	0	a
(a,d)	false	2	s
(b,a)	false	-1	b
(c,d)	false	2	s
(d,b)	false	-4	d
(d,e)	true	6	d
(e,b)	false	-4	d

Durchlauf 2:

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(v)	Vorgänger(v)
(s,a)	false	1	s
(s,c)	false	2	s
(s,d)	false	2	s
(a,b)	false	0	a
(a,d)	false	2	s
(b,a)	true	-2	b
(c,d)	false	2	s
(d,b)	false	-4	d
(d,e)	false	6	d
(e,b)	false	-4	d

Durchlauf 3:

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(v)	Vorgänger(v)
(s,a)	false	1	s
(s,c)	false	2	s
(s,d)	false	2	s
(a,b)	false	0	a
(a,d)	true	1	a
(b,a)	false	-2	b
(c,d)	false	2	s
(d,b)	true	-5	d
(d,e)	true	5	d
(e,b)	false	-5	d

Durchlauf 4:

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(v)	Vorgänger(v)
(s,a)	false	1	s
(s,c)	false	2	s
(s,d)	false	2	s
(a,b)	false	0	a
(a,d)	false	1	a
(b,a)	true	-3	b
(c,d)	false	2	s
(d,b)	false	-5	d
(d,e)	false	5	d
(e,b)	false	-5	d

Durchlauf 5:

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(v)	Vorgänger(v)
(s,a)	false	1	s
(s,c)	false	2	s
(s,d)	false	2	s
(a,b)	false	0	a
(a,d)	true	0	a
(b,a)	false	-3	b
(c,d)	false	2	s
(d,b)	true	-6	d
(d,e)	true	4	d
(e,b)	false	-6	d

Durchlauftest auf negativen Zykel

Kante(u,v)	(Distanz(u)+Gewicht(u,v) < Distanz(v))	Distanz(u)	Distanz(v)
(s,a)	false	0	-3
(s,c)	false	0	2
(s,d)	false	0	0
(a,b)	false	-3	-6
(a,d)	false	-3	0
(b,a)	true (⇒ break)	-6	-3
(c,d)		2	0
(d,b)		0	-6
(d,e)		0	4
(e,b)		4	-6

14.4 FLOYD/WARSHALL-ALGORITHMUS

Sei $G = (V, E)$ ein gerichteter Graph. Die *transitive Hülle* $G' = (V, E')$ des Graphen G, ist der Graph mit Schleifen, der folgendermaßen definiert ist:

$(u, v) \in E'$ gdw. $u = v$ oder es gibt einen Pfad von u nach v in G.

A) Berechnen Sie die transitive Hülle des folgenden Graphen.

$G' = (V, E')$ mit E'
$= \{(1,1),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)\}$

B) Modifizieren Sie den Algorithmus von Floyd/Warshall, um die transitive Hülle G' eines Graphen G in (n^3) zu berechnen. Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.
[H] A ist Adjazenzmatrix von G = (V,E)
$n = |V|$
baue neue Matrix(n x n) M
i= 1 to n j= 1 to n M[i][j] = A[i][j] i = 1 to n M[i][i] = 1 k= 1 to n i=1 to n j= 1 to n M[i][k] == 1
M[k][j] == 1 M[i][j] = 1 H

Korrektheit:
Jeder Knoten enthält eine Kante zu sich selbst (transitive Hülle), siehe Zeile 9-11. Außerdem übernimmt die transitive Hülle auch die bereits vorhandenen Kanten des Graphen. Der FW-Algorithmus prüft nun für jeden Knoten, ob die bisher berechnetete Distanz zwischen den Knotenpaaren kürzer wird. Da wir aber nur binär Denken müssen, es also nur 0 keinen Pfad gibt und 1 es gibt einen Pfad, entsteht nur eine neue Adjazenzmatrix.

Laufzeit:
ineinander geschachtelte For-Schleifen mit Laufzeit je $O(|V|)$, d.h. die Laufzeit des Algorithmus beträgt: $O(|V|^3)$

14.5 Programmieraufgabe: Dijkstra-Algorithmus

Laden Sie die Java-Vorlage aus dem Moodle herunter und machen Sie sich mit der Implementierung von Graphen vertraut, die Sie in der Vorlage finden. Implementieren Sie dann die Methode:

```
runDijkstra(Graph g, Node start),
```

die den Dijkstra-Algorithmus ausführen soll. Verwenden Sie bei Ihrer Implementierung sinnvolle Variablennamen und kommentieren Sie Ihren Code! Laden Sie Ihre Lösung ins Moodle. Nicht kompilierende Abgaben werden **mit 0 Punkten** bewertet.