ALGORITHMEN & DATENSTRUKTUREN

SKRIPT ZUR VORLESUNG SOSE 2015
UNIVERSITÄT TRIER

Autor: Prof. Stefan NÄHER
Digitalisierung: Thomas SCHIMPER

INHALTSVERZEICHNIS

1 DIVIDE & CONQUER

- Teile in K-Teilprobleme der Größe $n_1, \cdots, n_k$
- Beherrsche
- Zusammensetzen (Mischen)

1.1 LAUFZEIT

wird beschrieben durch eine rekursive (Un)gleichung

$$T(n) = \text{Laufzeit für Problem der Größe } n$$

$$T(n) = \underbrace{\sum_{i=1}^{k} T(n_i)}_{\text{Conquer}} + \underbrace{T_{\text{teile}}(n)}_{\text{Zeit zum Teilen}} + \underbrace{T_{\text{mischen}}(n)}_{\text{Zeit zum Mischen}}$$

Häufiger Fall: $k = 2$, $n_1 = n_2 = \frac{n}{2}$ dann $T(n) = 2 * T(\frac{n}{2}) + T_{\text{teile}}(n) + T_{\text{mische}}(n)$

1.2 BEISPIEL

MERGE-Sort (Sortieren durch Mischen)

PROBLEM Feld $A[1 \cdots n]$ von Zahlen

AUFGABE Permutiere die Eingabe von A so, dass gilt

$$A[i] \leq A[i+1] \text{ für } i = 1, \cdots, n-1$$

d.h.: aufsteigende Sortierung

IDEE VON MERGESORT Divide & Conquer

- Teile A in zwei gleich große Teilfelder. Dazu muss der Algorithmus (Mergesort) auf Teilprobleme angewendet werden.

```
Mergesort(l, r){
//sortiert das Teilfeld A[l, r] aufsteigend
if l ≥ r then
    return;
end
// Verankerung 0 oder 1. El
//Bei einem Felder der Laenge ≤ 1 nichts zu tun
m ← ⌊(l+r)/2⌋;
//Teileschritt
Mergesort(l, m);
Mergesort(m + 1, r);
Merge(l, m, r); //Mischen
}
```

Algorithm 1: Mergesort

```
//Mit Hilfsfeld B[1 ··· r − l + 1]
//Vorbedingung A[l ··· m] und A[m + 1 ··· r] sind sortiert
//Schritt 1: Mische die Zahlen in A[l ··· r]
//in eine sortierte Folde im Hilfsfeld B
for i = 1 to r-l+1 do
    A[l+i-1] ← B[i];
end
```

Algorithm 2: Merge

1.3 Laufzeitgleichung

Laufzeit Gleichung für Mergesort:

$$T(n) = 2 * T(\frac{n}{2}) + \underbrace{\mathcal{O}(1)}_{Teile} + \underbrace{\mathcal{O}(n)}_{Mische}$$
$$= 2 * T(\frac{n}{2}) + \mathcal{O}(n)$$

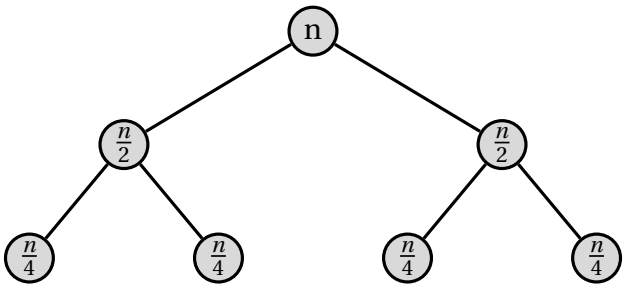
Genauer:

$$2 * T(\frac{n}{2}) + c * n \text{ , für eine Konstante } c > 0$$
$$\text{Für } n \leq 1 T(n) = \mathcal{O}(n)$$

1.4 Abschätzung rekursiver Laufzeitgleichungen

1.4.1 Substitutionsmethode

Rate die Lösungen und überprüfe die Korrektheit
Behauptung: $T(n) = \mathcal{O}(n * \log n)$. Genauer: $\exists$ Konstante c'. $T(n) \leq c' * n * \log n$ Veranschaulichung der Rekursion→ **Rekursionsbaum**:



Rekursionsbaum (Knoten=Teilprobleme)

Mergesort $T(n) \leq 2 * T(\frac{n}{2}) + c * n$ für eine Konstante c

Behauptung $T(n) \leq c' * n * \log n$ für eine Konstante $c' > c$

Beweis durch Induktion (Einsetzen → Substitution)

$$
\begin{aligned}
T(n) &\leq 2 * T(\frac{n}{2}) + c * n \\
&\quad Induktionsanfang \\
&\leq 2 * c' * \frac{n}{2} * \log(\frac{n}{2}) + c * n \\
&= c' * n * \log(n-1) + c * n \\
&= c' * n * \log n - c' * n + c * n \leq c' * n * \log(n) \text{ für } c' < c
\end{aligned}
$$

⇒ Behauptung □
Mergesort hat die Laufzeit $\mathcal{O}(n * \log n)$

1.4.2 Iterationsmethode

Beispiel

$$
\begin{aligned}
T(n) &= 3 * T(\frac{n}{4}) + n \\
T(n) &= 1, \text{ für } n \leq 1
\end{aligned}
$$

Iteriere die rekursive Gleichung bis zur Verankerung

$$
\begin{aligned}
T(n) &= 3 * T(\frac{n}{4}) + n \\
&= n + 3 * (\frac{n}{4} + 3 * T(\frac{n}{16})) \\
&= n + 3^1 * \frac{n}{4^1} + 3^2 * T(\frac{n}{4^2}) \\
&= n + 3^1 * \frac{n}{4^1} + 3^2 * \frac{n}{4^2} + 3^3 * T(\frac{n}{4^3}) \\
&\dots \\
&= n + \sum_{i=1}^{k-1} ((\frac{3}{4})^i) * n + 3^k * \underbrace{T(\frac{n}{4^k})}_{\text{Verankerung}} \\
&\text{für } k = \log_4 n \text{ gilt } \frac{n}{4^k} \leq 1. \\
&\text{Dann gilt } T(\frac{n}{4^k}) = 1 \\
&= n + \sum_{i=1}^{\log_4 n - 1} (\frac{3}{4})^i * n + 3^{\log_4 n} \\
3 * \log_4 n \leq 4^{\log_4 n} &= n \\
\Rightarrow T(n) &\leq \sum_{i=0}^{\infty} (\frac{3}{4})^i * n + n \\
&= n * \sum_{i=0}^{\infty} (\frac{3}{4})^i + n \\
&\leq 4n + n = 4n \Rightarrow \underline{T(n) = \mathcal{O}(n)}
\end{aligned}
$$

ALLGEMEINER

$$T(n) = a * T(\frac{n}{b}) + f(n) (\rightarrow \text{Master-Lemma})$$

2 DIE WICHTIGSTEN SUMMENFORMELN

2.1 ARITHMETISCHE REIHE

$$\begin{aligned}\sum_{k=1}^{n}(k = 1 + \cdots + n) &= \frac{1}{2} * n * (n+1) \\ &= \frac{1}{2} * (n^2 + n) \\ &= \mathcal{O}(n^2)\end{aligned}$$

2.2 GEOMETRISCHE REIHE

2.2.1 VERSION A

$$\sum_{k=0}^{n}(x^k) = \frac{x^{n+1}-1}{x-1}\text{ , für } x \neq 1$$

2.2.2 VERSION B

$$\sum_{k=0}^{n}(x^k) = \frac{1}{1-x}\text{ , für } |x| \leq 1$$

2.3 HARMONISCHE REIHE

$$H_n : \sum_{k=1}^{n}(\frac{1}{k}) \leq 1 + \ln(n), n\text{-te harmonische Zahl} = \mathcal{O}(\log n)$$

2.4 INTEGRIERENDE REIHE

$$\sum_{k=0}^{\infty}(k * x^k) = \frac{x}{(1-x)^2}\text{, für } |x| < 1$$

2.5 TELESKOPSUMMEN

$$\text{Folge } a_0, a_1, \cdots, a_n$$

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0$$

3 EINFACHE DATENSTRUKTUREN

Keller(Stack),Schlange(Queue),Listen(Feld/Array)

3.1 KELLER

Keller oder Stack (Stapel) Zugriff nur auf das oberste Eement

3.1.1 EINGESCHAFTEN

- beschränkt, d.h. maximale Größe n

DATENSTRUKTUR Feld $A[1 \cdots n]$
Index (int) top
top = 0 $\Leftrightarrow$ Stack ist leer

OPERATIONEN AUF EINEN STACK S

- s.clear() = top$\leftarrow$ 0;
- s.empty() = return top = 0;
- s.push(x) = top $\leftarrow$ top++;
  A[top] $\leftarrow x$;
- s.pop() = return A[top−−];
- s.top() = return A[top];

Alle Operationen haben Laufzeit $\mathcal{O}(1)$

PLATZBEDARF $n+1$ Speicherzellen = $\mathcal{O}(n)$

EXCEPTIONS Overflow & Underflow

- unbeschränkt : beliebige Größe (dynamisch) → später (Listen)

3.2 SCHLANGE

3.2.1 FIFO-QUEUE (FIRST IN FIRST OUT)

- beschränkt, d.h. maximale Größe (**Kapazität**) n

  FELD $A[1 \cdots n]$

  2 INDICES $first$ & $stop$

  OPERATIONEN AUF EINE QUEUE Q

  - Append(PushBack) = A[stop++]$\leftarrow x$;
  - Pop(PopFront) = return $A[first++]$;
  - Q.clear() = $first \leftarrow 1$;
    $stop \leftarrow 1$;
  - Q.empty() = return $first = stop$;
  - Q.append(x) = $A[stop] \leftarrow x$;
    $if\ ++stop = n+2$ then $stop \leftarrow 1\ fi$;
    (Alternativ:(Modulo))
  - Q.pop(x) = $x \leftarrow A[first]$;
    $if\ ++first = n+2$ then $first \leftarrow 1\ fi$;
    return $A[first]$;

    ACHTUNG Over-/Underflow

    KOSTEN Laufzeit $\mathcal{O}(1)$

  PROBLEM Queue-Elemente $A[first \cdots stop-1]$ wandern
  nach rechts $\Rightarrow stop > n$

  LÖSUNG „Zirkuläre“ Feld $A[1 \cdots n+1]$
  ($n+1$ zur Unterscheidung zwischen voll und leer)

- unbeschränkt $\Rightarrow$ später (Listen)

3.3 Listen

- Einfach verkette Listen

IDEE jedes Element merkt sich (Referenz/Pointer/Adresse), wo sein Nachfolger(next) im Speicher steht

SYMBOL FÜR LISTENELEMENTE

BEISPIEL FÜR LISTEN Graphik einfügen :)

4 Heapsort

Sortieren durch Max-Auswahl (Grafik einfügen)

LINEARE SUCHE nach Maximum in Teilfeldern $A[1 \cdots r]$ für $r = n, \cdots, 1$

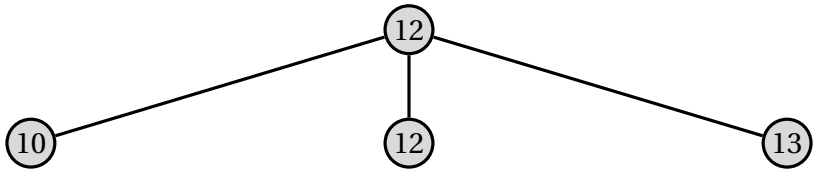
FRAGE Kann man das Maximum schneller finden? (Ausnutzung aller Vergleiche)

ANTWORT Ja $\Rightarrow$ Datenstruktur : $Heap$

4.1 Definition Maxheap

Ein Heap ist ein Baum, dessen Knoten mit Zahlen beschriftet sind, sodass für alle Knoten v gilt (außer Wurel):

$$Zahl(v) \leq Zahl(Vater(v))$$



Blätter sind Knoten ohne Kinder!

BEDEUTUNG Wie viel enthält das Maximum !
Im Heapsort verwenden wir ausgeglichene binäre Heaps

BINÄR Alle Knoten haben 0 oder 2 Kinder bis eventuell auf einen Knote mit einem Kind

AUSGEGLICHEN Es gibt ein $K \geq 0$, sodass gilt:

alle Blätter haben Tiefe K oder $K+1$

Auf Tiefe $K+1$ stehen Blätter Möglichst weit links

BEOBACHTUNG Ausgeglichene binäre Heaps lassen sich sehr kompakt als Feld darstellen (Level für Level)

Dann gilt:

$$\text{Kinder von } A[i] \text{ sind } A[2-i] \text{ und } A[2*i+1]$$
$$\text{Vater von } A[i] \text{ ist } A[\lfloor \frac{i}{2} \rfloor] \text{ für } i > 1$$
$$A[i] \text{ ist Blatt} \Leftrightarrow 2*i > n$$

4.2 DEFINITION

Ein Feld $A[1 \cdots n]$ heißt HEAP, falls $A[\lfloor \frac{i}{2} \rfloor] > A[i]$ für $2 \leq i \leq n$

ANWENDUNG AUF SORTIERUNG

4.2.1 AUFBAUPHASE

Verwende $A[1 \cdots n]$ in einem Heap (s. Definition)
$\Rightarrow$ Maximum in $A[1]$ (Wurzel)

4.2.2 SELEKTIONSPHASE

```
for r = n downto 1 do
    A[1] ↔ A[r];
    Verwandle A in Heap;
end
```

4.3 REALISIERUNG

4.3.1 AUFBAUPHASE

Aufbau durch „Heruntersinken lassen“→ SINK (→ auch heapify)
SINK(i) := lasse A[i] heruntersinken

- Vertausche $A[i]$ mit dem Maximum seiner Kinder.
  Sei $A[j]$ dieses Maximum $j \in \{2i, 2i+1\}$
- Setze Heruntersinken mit $A[i]$ fort (SINK(j))
- Wiederhole bis entweder Blatt erreicht oder Heapeigenschaft erreicht

Sink(i) wird nur ausgeführt, falls Heapeigenschaft verletzt. Wir bauen von unten nach oben immer höhere Teil-Heaps

BEOBACHTUNG Jedes Blatt ist für sich ein Heap der Höhe 0.

Lasse zunächst die Väter der Blätter sinken → Heaps der Höhe 1 dann deren Väter → Höhe 2 … bis zur Wurzel

```
for i = ⌊n/2⌋ downto 1 do
  | SINK(i, n);
end
```

4.3.2 IMPLEMENTIERUNG VON SINK ITERATIV

```
SINK(i, r){
        //lasse A[i] Teilfeld A[1 ··· r] sinken
        x ← A[i];
        j ← 2 ∗ i;          //linkes Kind
while j ≤ r do
    | if j + 1 ≤ r then
    |   | if A[j + 1] > A[j] then
    |   |   | j ← j + 1;          //rechtes Kind Größer
    |   | end
    | end
    | if x ≥ A[j] then
    |   | break;          //Heap ist in Ordnung
    | end
    | A[i] ← A[j]          //Hochkopieren des größten Kindes
    | i ← j          //Setze an der Stelle j fort
    | j ← 2 ∗ i;
end
        A[i] ← x;
}
```

4.3.3 SELEKTIONSPHASE

```
r ← n;
while r > 1 do
  | A[1] ↔ A[r];
  | r ← r − 1;
  | SINK(1, r);
end
```

4.4 Laufzeitanalyse

4.4.1 Beobachtung

Ein Aufruf SINK(i,n) hat Laufzeit $\mathcal{O}$(„Höhe von i im Heap“)
$\Rightarrow$ worst-case: Also ist die Gesamtlaufzeit der Aufbauphase

$$\begin{aligned}\mathcal{O}(\sum_{i=1}^{\frac{n}{2}} \text{Höhe}(i)) &= \mathcal{O}(\sum_{n=1}^{\log n} \underbrace{(h * \#\text{Knoten auf Höhe} h))}_{\leq \frac{n}{n^2}} \\ &= \mathcal{O}(\sum_{n=0}^{\infty} (h * \frac{n}{2^n})) \\ &= \mathcal{O}(n * \sum_{n=0}^{\infty} (\frac{n}{2^n})) \\ &= \mathcal{O}(n)\end{aligned}$$

INTEGRIERENDE REIHE

$$\begin{aligned}\sum_{n=0}^{\infty} (h * x^n) &= \frac{x}{(1-x)^2}, \text{ falls } x < 1 \\ \sum_{n=0}^{\infty} (\frac{n}{2^n}) &= \sum_{n=0}^{\infty} (h * (\frac{1}{2})^n) \text{ hier } x = \frac{1}{2} \\ &= \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2\end{aligned}$$

$\Rightarrow$ Aufbauphase hat Laufzeit $\mathcal{O}(n)$

4.4.2 Laufzeitanalyse der Selektionsphase

$$\begin{aligned}\mathcal{O}(\sum_{r=1}^{n} \underbrace{Kosten\ von\ Sink(1,r))}_{\leq \text{Höhe des Heaps} A[1 \cdots r]} \\ \leq \log r \\ \mathcal{O}(\sum_{r=1}^{n} (\log r)) \leq \mathcal{O}(\sum_{r=1}^{1} (\log n)) \\ = \mathcal{O}(n * \log n)\end{aligned}$$

SATZ Auf einem Feld der Länge n hat HEAPSORT eine Laufzeit $\mathcal{O}(n * \log n)$

BEMERKUNG

- Feld in Heap verwandeln braucht nur lineare Laufzeit $\mathcal{O}(n)$
- Selektionsphase dominiert (quadratisch)
- Heapsort braucht einen zusätzlichen Platz($\leftrightarrow$ Mergesort) d.h. läuft nur auf Eingabefeld A
  $\Rightarrow$ **in-place-Eigenschaft**

- In der Praxis ist Heapsort nicht sehr effizient.
  Grund: Speicherzugriffe haben schlechte Lokalität → (Cache-Fehler)
  Besser: Divide & Conquer

5 QUICKSORT

Gilt in der Praxis als schnellstes Sortierverfahren

DIVIDE & CONQUER Arbeit wird im Teilschritt gemacht (↔ Mergesort: Mischen)

```
QUICKSORT(l, r){
if l ≥ r then
  | return;          //Verankerung
end
//Partition
x ← A[l];            //Pivotelement
i ← l + 1;
j ← r;
repeat
  | while i ≤ r ∧ A[i] < x do
  |   | i++;
  | end
  | while j ≥ l + 1 ∧ A[j] ≥ x do
  |   | j − −;
  | end
  | if i < j then
  |   | A[i] ↔ A[j];
  | end
until i > j;
A[l] ↔ A[j];         //bringt x an korrekte Position
QUICKSORT(l, j − 1);  //Conquer
QUICKSORT(j + 1, r);  //Conquer
}
```

5.1 ANALYSE VON QUICKSORT

Laufzeit der Partitionierung ist linear, d.h. $\mathcal{O}(n)$

BEOBACHTUNG Laufzeit = $\mathcal{O}$(# Vergleiche)

5.2 KOSTENANALYSE

5.2.1 KOSTEN IM SCHLECHTESTEN FALL

Aufruf von Quicksort(l,r)

- Partitionierung : $\mathcal{O}(r-l+1)$
- Kosten der rekursiven Aufrufe

Sei $QS(n)$ = maximale Zahl von Vergleichen, die Quicksort auf das Feld der Längen ausgeführt
→ Rekursionsgleichung (# Vergleiche)

$$QS(n) = n + \max\{QS(j-1), QS(n-j)\}$$

SCHLECHTESTER FALL Position jedes Pivotelements ist extrem, d.h $j = 1 \vee j = n$

BESTER FALL Pivotelement kommt in die Mitte

$$\begin{aligned} &QS(\widetilde{n}) \\ &\quad = QS(\frac{n}{2} * 2 + n \\ &\quad = \mathcal{O}(n * \log n) \text{ (s. Mergesort)} \end{aligned}$$

⇒ arithmetische Reihe, d.h. $QS(n) = \frac{1}{2} * n * (n-1) = \mathcal{O}(n^2)$

MÖGLICHE EINGABE FÜR DEN WORST-CASE sortiertes Array

5.2.2 ERWARTETE KOSTEN VON QUICKSORT (MITTLERE LAUFZEIT)

d.h. *Erwartungswert* für # Vergleiche bei einer zufälligen Eingabe:

ANNAHMEN

- alle Zahlen im Feld $A[i \cdots n]$ sind paarweise verschieden
- jede der $n!$ möglichen Permutationen der Eingabe sind gleich wahrscheinlich

Wir können ohne Beschränkung der Allgemeinheit anrechen, dass die Werte die Zahlen $1, \cdots, n$ sind und das Pivotelement $A[1] = k$ mit Wahrscheinlichkeit $\frac{1}{n} \forall 1 \leq k \leq n$. Dann müssen rekursive Teilprobleme der Größe $k-1$ und $n-k$ gelöst werden. Diese sind wieder zufällig Folgen, d.h. sie erfüllen die obigen Annahmen. Sei nun :

$$\begin{aligned} \overline{QS}(n) :=& \text{erwartete (oder mittlere) Anzahl} \\ & \text{von Vergleichen auf einem Feld der Länge n} \end{aligned}$$

Wir wissen $prob(A[i] = k) = \frac{1}{n}$ für $1 \leq k \leq n$

ERWARTUNGSWERT $\sum$(„Wahrscheinlichkeit des Falls“)*(„Wert des Falls“))

bei Gleichverteilung (alle n Fälle haben $prob(\frac{1}{n})$) $\Rightarrow \frac{1}{n} * \sum_{k=1}^{n}$(Werte)
Dann gilt:

$$\overline{QS}(0) = \overline{QS}(1) = 0 \text{ (Kein Vergleich)}$$

$$\text{Für } n \geq 1:$$

$$\overline{QS}(n) \leq n + \underbrace{E(\overbrace{\overline{QS}(A) + \overline{QS}(B}^{\text{Teilprobleme}})}_{\sum_{k=1}^{n}(\frac{1}{n} * (\overline{QS}(k-1) + \overline{QS}(n-k)))}$$

$$\overline{QS}(n) \leq n + \frac{1}{n} \sum_{k-1}^{n} (\overline{QS}(k-1) + \overline{QS}(n-k)$$

$$\leq n + \frac{2}{n} * \sum_{k=0}^{n-1} (\overline{QS}(K)$$

$$n * \overline{QS}(n) \leq n^2 + 2 * \sum_{k=0}^{n-1} (\overline{QS}(K)) \qquad (*)$$

$$(n+1) * \overline{QS}(n+1) * (n+1)^2 + 2 * \sum_{k=0}^{n} (\overline{QS}(K)) \qquad (**)$$

$$(n+1) * \overline{QS}(n+1) = n * \overline{QS}(n) \qquad ((*)*(**))$$

$$\leq (n+1)^2 - n^2 + 2 * \overline{QS}(n)$$

$$(n+1) * \overline{QS}(n+1) \leq 2n + 1 + (n+2) * \overline{QS}(n)$$

$$\overline{QS}(n+1) \leq \frac{n+2}{n+1} * \overline{QS}(n)$$

$$= 2 + \frac{n+2}{n+1} * (2 + \frac{n+1}{n} * (2 + \frac{n}{n-1} \cdots)$$

$$= 2 * (n+2) * (\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + 1)$$

$$\Rightarrow \overline{QS}(n) \leq 2 * (1 + (n+1) * \underbrace{\sum_{i=1}^{n} (\frac{1}{i})}_{\text{harmonische Reihe}} \leq 1 + \ln(n)$$

$= \mathcal{O}(n * \ln(n)) = \mathcal{O}(n * \log(n)) \Rightarrow$ erwartete Laufzeit $QS * \mathcal{O}(n * \log(n))$

FRAGE Wie kann man die schlechte Laufzeit vermeiden ?
→ Randomisiertes Quicksort

- Permutiere Eingabe zufällig
  ⇒ Annahmen sind erfüllt ✓
- Zufällige Wahl des Pivotelements am Anfang der Randomisierung

6 Allgemeine Sortierverfahren

- Mergesort
- Heapsort in-place
- Quicksort in-place *sehr schnell*

7 Spezielle Sortieralgorithmen

allgemeine Sortieralgorithmen verwenden Vergleiche und haben Laufzeit

$$\mathcal{O}(n * \log n)$$

In speziellen Situationen kann man schneller sortieren ($\rightarrow \mathcal{O}(n)$)

Hier Schlüssel sind ganze Zahlen aus Intervall $\{1, \cdots, k\}(0, \cdots, k-1)\,k$ ist konstant

Beispiele

- sortiere Briefe nach PLZ ($k < 10^6$)
- Studierende nach Martrikelnummer
- Strings (Namen) nach dem 1.Buchstaben (ASCI $k = 256$)
- ...
- Farben

Vereinfachung Sortiere n Schlüssel aus $\{1, \cdots, k\}$

Problem Eingabefeld $A[1, \cdots, n]$ mit $A[i] \in \{1, \cdots, \text{Ⓚ}\}$

Ausgabe aufsteigend sortiertes Feld $\rightarrow B[1, \cdots, n]$

7.1 COUNTINGSORT

7.1.1 ALGORITHMUS

```
COUNTINGSORT{
3 Felder :      A[1,···,n] //Eingabe
                B[1,···,n] //Ergebnis
                C[1,···,k] //Hilfsfeld
Schritt 1:
//Zähle, wie oft jedes x ∈ A vorkommt
for i = 1 to k do
  | C[i] ← 0;
end
for j = 1 to n do
  | x ← A[j];
  | C[x] + +;
end
Schritt 2:
//Rechne für jeden Schlüssel x ∈ A aus, wo er in der sortierten Folge
//(Index im Feld B) stehen soll
pos = ∑_{i=1}^{x} C[i];
for i = 2 to do
  | C[i]+ = C[i − 1];
end
Schritt 3:
//Ausgabe in Feld B
for j = n downto 1 do
  | x ← A[j];
  | B[C[x]] ← x;
  | C[x] − −;
end
}
```

BEMERKUNG

- Laufe das Feld A rückwärts durch ($\rightarrow$ Stabilität)
- In jedem Schritt schreibe x möglichst weit rechts nach B (Position = $C[x]$;)
- Schreibe nächstes x links daneben ($\rightarrow C[x] + +$;)

LAUFZEITANALYSE

- 4 For-Schleifen mit konstanter Laufzeit im Rumpf
- Iterationen über $C \rightarrow \mathcal{O}(k)$
- ———— über $A \rightarrow \mathcal{O}(n)$

GESAMTLAUFZEIT $\mathcal{O}(n+k)$

NACHTEIL zusätzlicher Speicherplatz $n+k$ für B&C, die nicht *in-place*

7.2 BUCKETSORT

7.2.1 DEFINITION

Sortieren durch Fachverteilung

IDEE Verteile die Schlüssel aus A auf k Buckets (Körbe,Fächer).
$B[1,\cdots,n]$ ist Feld von n Listen

```
foreach x ∈ A do
    for i = 1 to n do
        x ← A[i];
        B[x].append(x);
    end
end
```

AUFSAMMELN Durchlaufe alle Listen (Buckets) $B[i]$ und gebe die Elemente aus

LAUFZEIT

- Initialisierung : k leere Listen im Feld $B[1,\cdots,k] = \mathcal{O}(n)$
- Verteilung: $n\times$ Einfügen $(append) = \mathcal{O}(n)$
- Aufsammeln *ggf:* $\mathcal{O}(k)$?

$\Rightarrow \mathcal{O}(n+k) = \mathcal{O}(n)$,falls $k = \mathcal{O}(n)$

8 DATENSTRUKTUREN FÜR MENGEN

8.1 STATISCHE DATENSTRUKTUREN

SITUATION Menge S von n Datensätzen (Objekte), jeder Datensatz besitzt einen Schlüssel. Wir möchten S in einer Datenstruktur D speichern, die folgende Operationen effizient unterstützt:

OPERATIONEN

- D.insert($key, date$) : Fügt ein neues Objekt mit Schlüsselkey und Daten $data$ zu S hinzu. Falls key bereits vorhanden → überschreibe
- D.delete(key): Entferne das Objekt mit Schlüssel key, falls vorhanden

VARIANTEN VON LOOKUPS

- nur Test → *boolean* (true oder false)
- gib Objekt zurück (*null* , falls nicht vorhanden) dann eventueller Zugriff auf die Daten(z.B. Telefonnummern)

ANWENDUNGEN Beispiel: Martrikelnummer,Name,Id, $\cdots$ und eventuell weitere Namen.

AB JETZT Wir betrachten wir nur noch Schlüssel und diese sind ganze Zahlen

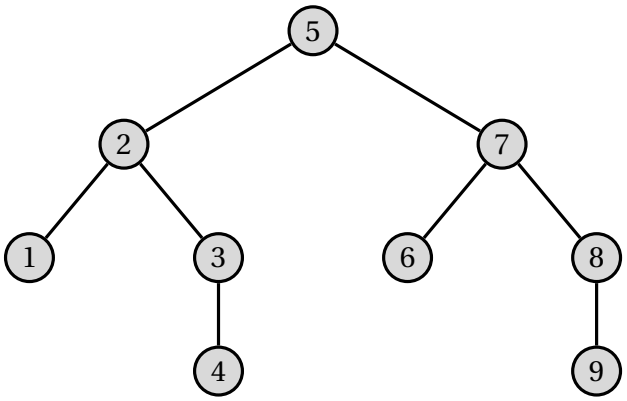
8.2 DYNAMISCHE DATENSTRUKTUREN

- Knoten-orientierte binäre Suchbäume (später blatt-orientiert)

IDEE S wird in den Knoten eines binären Baums gespeichert. Die binäre Suche definiert *implizit* einen solchen Baum.

BEISPIEL $1,2,3,4,\underbrace{\textcircled{5}}_{m},6,7,8,9$

BAUM



repräsentiert alle möglichen Abläufe der
binäre Suche nach x

Jeder konkrete Ablauf nach einen Schlüssel x entspricht einem Pfad im Baum der Wurzel

- bis zu einem Knoten (bei erfolgreicher Suche)
- bis zum $Null$-Verweis (falls $x \notin S$)

8.2.1 EXPLIZITER AUFBAU EINES BINÄREN SUCHBAUMES

Sei $A[1 \cdots n]$ aufsteigend sortiertes Feld

REKURSIVE KONSTRUKTION Ein binärere Suchbaum T für $A[1 \cdots n]$ besteht aus

- Wurzelknoten v mit Schlüssel $kx = A[m]$, wobei $m = \lfloor \frac{n+1}{2} \rfloor$
- ein binärer Suchbaum T' für $A[1 \cdots m-1]$ als *linken* Unterbaum von v
- ——————————— T'' für $A[m+1 \cdots n]$ als *rechten* Unterbaum von v

VERANKERUNG binärer Suchbaum für leere Mengen ist leerer Baum,
d.h. $null$-Referenz

REKURSIVE FUNKTION BaueBaum(l,r) konstruiert einen binären Suchbaum für das Teilfeld $A[l \cdots r]$

STRUKTUR zur Darstellung der Knoten

```
class bintree_node{
int key;
bintree_node left;
bintree_node right;
bintree_node parent;
}
```

BAUEBAUM(L,R)

- gibt einen Verweis auf die Wurzel zurück
- konstruiert den Baum für das Teilfeld $A[l \cdots r]$
- Eingabe: sortiertes Feld $A[1 \cdots n]$
- Verankerung: leeres Teilfeld

```
bintree_node BaueBaum(A, l, r){
if l > r then
  | return 0;
end
m ← ⌊(l+r)/2⌋;
p ← new bintree_node();
p.key ← A[m];
p.left ← BaueBaum(l, m−1);
p.right ← BaueBaum(m+1, r);
return p;
}
```

```
class bintree{
bintree_node root;
root ← BaueBaum(A, 1, n);
}
```

8.2.2 OPERATIONEN

- lookup(x) startet in der Wurzel, durchläuft einen Pfad nach unten bis entweder ein Knoten mit Schlüssel x gefunden oder endet in einer $null$-Referenz

```
bintree_node lookup(x);
p ← root;
while p ≠ O do
    if x = p.key then
        break;
    end
    if x < p.key then
        p ← p.left
    else
        p ← p.right;
    end
end
return p;
}
```

LAUFZEIT VON LOOKUP(X) $\mathcal{O}$(Höhe(T))
bei perfekt balancierten Bäumen $\Rightarrow \mathcal{O}(\log n)$

Bei Updates ($insert$ & $delete$) können Bäume schlecht balanciert sein (eventl. degeneriert)

VARIANTE VON LOOKUP(X) $\rightarrow locate(x)$

bei erfolgloser Suche: letzter Knoten$\neq 0$

- insert(x)

ANNAHME $x \notin T$ ($\rightarrow lookup(x) = null$)

Einmal füllen bitte

- delete(x)

ANNAHME $x \in T$

Suche endet in einem Knoten v mit $v.key = x$ $(lookup)$

1.FALL v ist Blatt (d.h. $v.left = v.right = null$) $\Rightarrow$ entferne v aus T

Sei p der Vater von x

```
if v = p.left then
  | p.left ← null;
else
  | p.right ← null;
end
```

2.FALL v hat genau ein Kind w
$\Rightarrow$ Ersetze v durch w

```
if v = p.left then
  | p.left ← w;
  | p.right ← w;
end
```

Lokale Situation gleicht einer Liste

3.FALL v hat zwei Kinder

ersetze $v.key$ durch Maximum um linken Unterbaum. Den Knoten u mit $u.keyMax$ finden wir wie folgt:

```
u = v.left
while u.right ≠ null do
  | u ← u.right;
end
//Kopiere u.key nach v
v.key ← u.key;
```

8.3 Laufzeit von delete(x) und insert(x)

$\mathcal{O}(\text{Höhe}(T))$✓

PROBLEM Nach einer Folge von Updates gilt nicht mehr, dass die Höhe= $\mathcal{O}(\log n)$

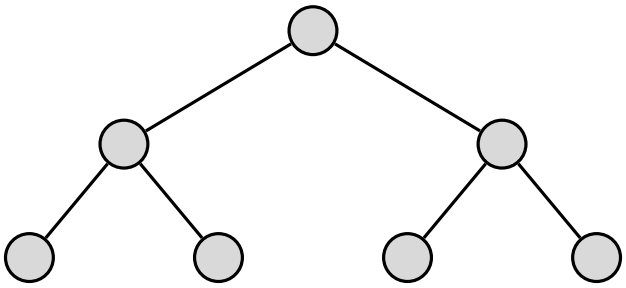
9 Blattorientierte binäre Bäume

9.1 Definition

Schlüssel werden von links nach rechts aufsteigend sortiert in den Blättern eines binären Baumes abgespeichert. In den inneren Knoten werden Wegweiser gespeichert (für die Suche).

genauer: Ein innerer Knoten v enthält einen Wert x, sodass alle Schlüssel im inneren (rechten) Unterbaum $\leq x$ ($> x$)

9.2 Beispiel



$S = \{2,5,9,20,27,30,37\}$

9.2.1 Beobachtungen

- Als Wegweiser kommen immer die maximalen Schlüssel um *linken* Unterbaum in Frage
- Die Menge der Schlüssel kann leicht als Liste realisiert werden (Verkettung der Blätter)

9.3 AUFBAU

durch eine rekursive Funktion!
$BaueBaum(A,l,r)$ konstruirt einen blattorientierten binären Baum

9.4 OPERATIONEN

9.4.1 LOOKUP(X)

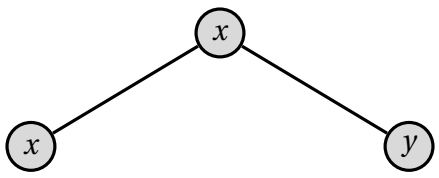
```
//liefert Blatt mit Inhalt x oder null p = root;
if p=null then
    return null;
end
while p ist kein Blatt do
    if x ≤ p.key then
        p ← p.left;
    else
        p ← p.right;
    end
end
if p.key = x then
    return p;
else
    return null;
end
```

9.4.2 INSERT(X)

Lookup endet in einem Blatt v mit Inhalt

$$y \qquad x \neq y$$

Dann ersetze das Blatt v durch

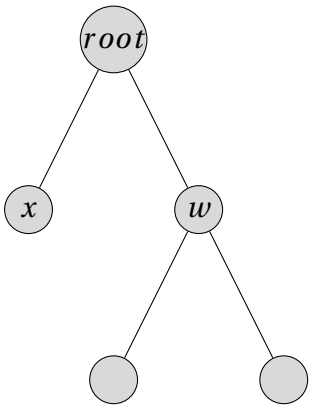


9.4.3 DELETE(X)

$x \in S$

Lookup(x) liefert das Blatt v mit Inhalt x

Sei w der Geschwisterknoten von v.



AKTION ersetze den Vater von v durch w

SYMMETRISCHE FÄLLE & SONDERFÄLLE

- Wurzel ändert sich
- Baum wird leer

Für beide Varianten gilt

- $lookup, insert, delete$ durchlaufen den Pfad des Baums herunter und führen eventuell einige lokale Änderungen aus.
- Baum T heißt balanciert, wenn die Höhe(T) = $\mathcal{O}(\log n)$, sonst unbalanciert (eventuell degeneriert)

IDEE Versuche den Baum nach jeder Updateoperation durch lokale Modifikation balanciert zu halten z.B. so, dass Höhe(T) ≤ 2 ($\log n \rightarrow$ perfect balanciert)

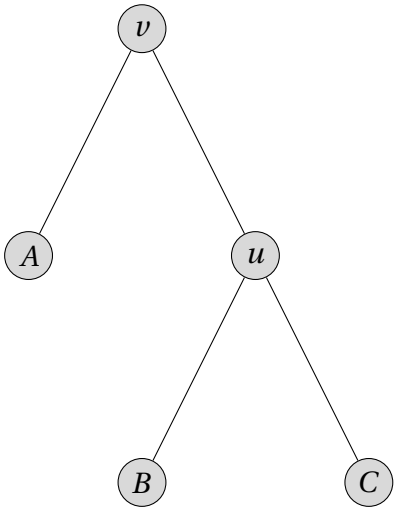
WICHTIG Laufzeit soll $\mathcal{O}$(Höhe(T)) bleiben!

9.5 LOKALE MODIFIKATIONEN

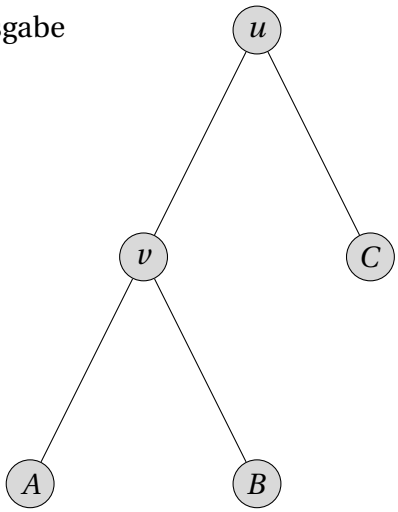
9.5.1 ROTATIONEN

ROTATION NACH LINKS

Eingabe

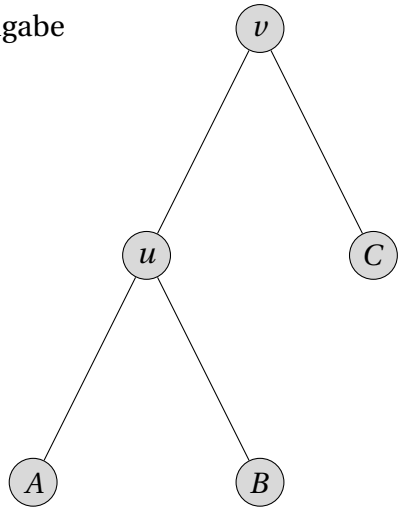


Ausgabe

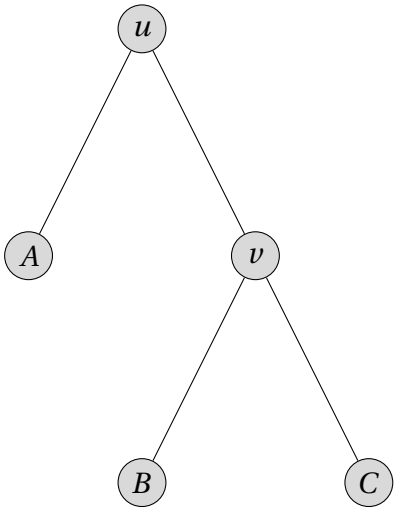


ROTATION NACH RECHTS

Eingabe



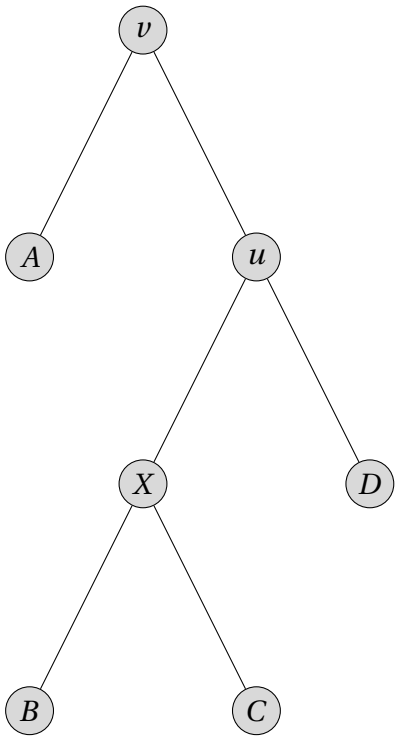
Ausgabe



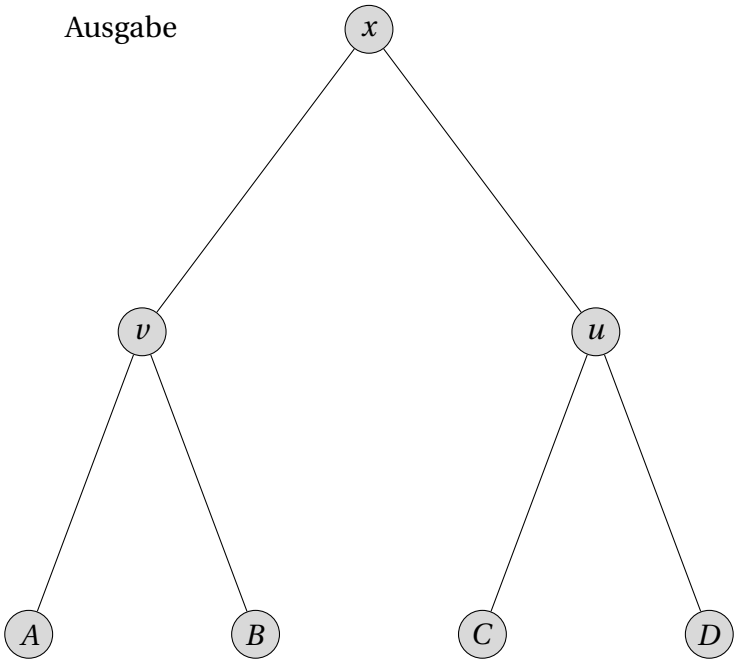
9.5.2 DOPPELROTATIONEN

NACH LINKS

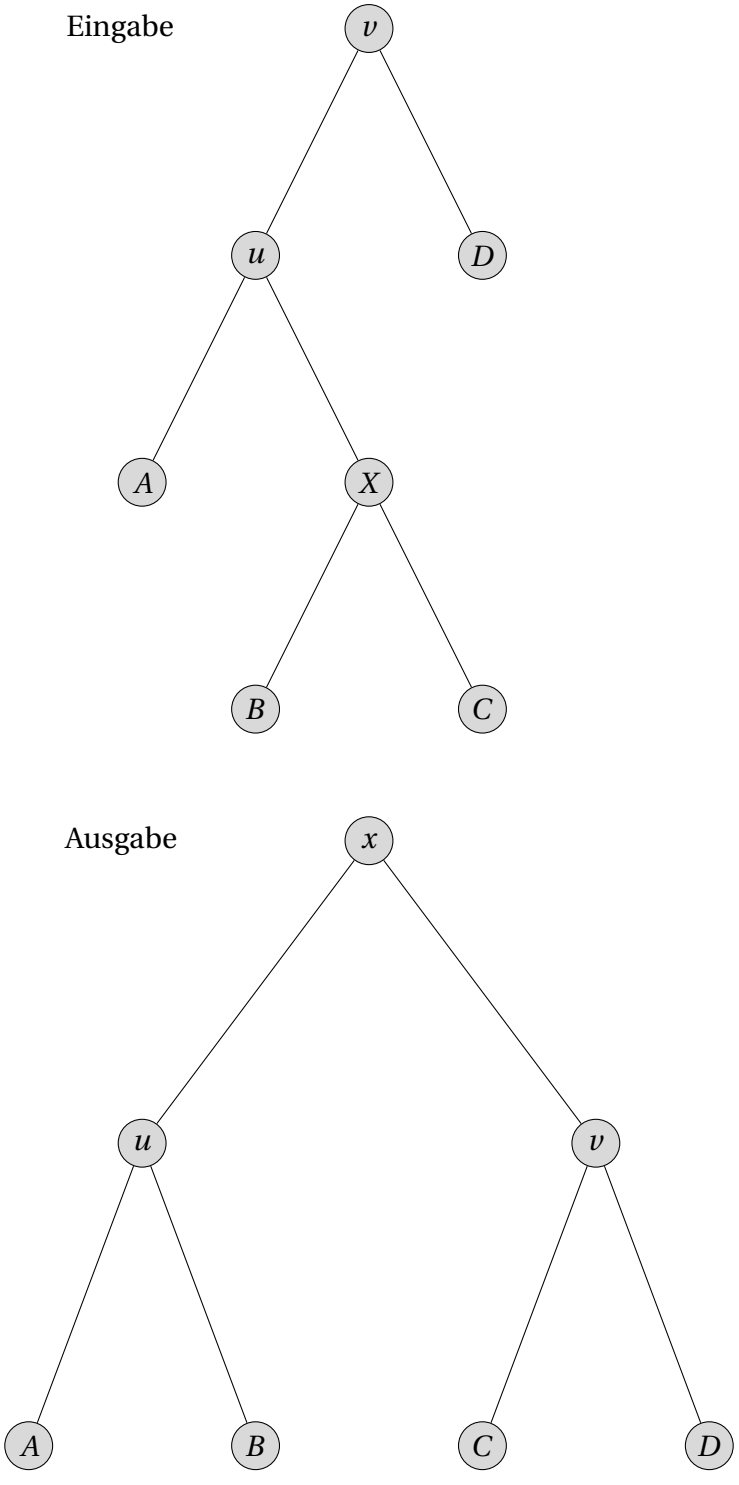
Eingabe



Ausgabe



NACH RECHTS



9.5.3 BEOBACHTUNG

$$double\_rotate\_left(x) = rotage\_right(w) + rotate\_left(v);$$
$$double\_rotate\_right(x) = rotate\_left(v) + rotate\_right(w);$$

10 BALANCIERTE BINÄRE BÄUME

10.1 IDEE

Verwendung von Rotation

10.2 GRUNDLEGENDE STRATEGIE

10.2.1 GEWICHTSBALANCIERTE BÄUME

$$Gewicht(T) = Anzahl\ Knoten\ in\ T$$

BALANCE-KRITERIUM $\forall V(v)$ gilt: $\frac{Gewicht(T_l(r))}{Gewicht(T_r(r))}$

10.3 AVL-BÄUME

10.3.1 LEMMA I

Ein AVL-Baum kann nach einer Insert-Operation durch Rotation/Doppelrotation rebalanciert werden

10.3.2 LEMMA II

Nach Delete durch Folge von Rotationen/Doppelrotationen entlang des Suchpfades (worst-case)

10.3.3 ZUSAMMENFASSUNG

Nach Update kann die ALL-Eigenschaft in Zeit $\mathcal{O}$(Höhe(T)) hergestellt werden

FRAGE Wie groß kann Höhe(T) im All-Baum m sein ($2 \geq 2^n$)

ZIEL $\mathcal{O}(\log n)$

10.3.4 ANALYSE

DEFINITION Sei $N(n)$ die Mindestanzahl von Knoten in einem All-Baum der Höhe h

DANN GILT

$$N(0) = 1$$
$$N(1) = 2$$
$$N(n) = 1 + N(n-2) + N(n-1)$$

BEOBACHTUNG erinnert an die **Fibonacci-Folge**

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-2} + F_{k-1}$$

HIER

$$N(k) \stackrel{!}{=} F_{k+3} - 1$$

BEWEIS

$$\begin{aligned} &Induktionsanfang: \\ &\qquad N(n) = F_{k+3} - 1 \\ &Induktionsschritt \\ &N(k+1) = 1 + N(n+1) \\ &\qquad = 1 + F_{k+2} - 1 + F_{k+3} - 1 \\ &\qquad = 1 + F_{k+4} \checkmark \end{aligned}$$

$\Rightarrow$ Die Mindestanzahl $N(n)$ von Knoten in einem AVL-Baum der Höhe h ist $F_{n+3} - 1$. Sei nun n die Anzahl der Knoten in einem Baum der Höhe h

$\Rightarrow$

$$\begin{aligned} n &\geq F_{n+3} - 1 \\ n + 1 &\geq F_{n+3} \end{aligned}$$

MAN WEISS

$$F_k \geq \frac{1}{\sqrt{5}} * \underbrace{(\frac{1+\sqrt{5}}{2})^k}_{\Phi \approx 1{,}618}$$

$$\Rightarrow \frac{1}{\sqrt{5}} \Phi \leq n + 1 \qquad | \log_\Phi$$

$$\Rightarrow \log_\Phi (\frac{1}{\sqrt{5}} + (n+3) \leq \log_\Phi (n+1)$$

$$\Rightarrow h \leq 1{,}44 * \log n$$

SATZ AVL-Bäume unterstützen die Wörterbuchoperationen INSERT,DELETE & LOOKUP auf eine Menge von n Schlüsseln in Zeit $\mathcal{O}(\log n)$ und Platz $\mathcal{O}(n)$.

10.3.5 BEMERKUNG

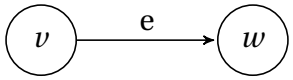
- Es gibt eine Reihe von anderen Balancierungstechniken, die aber ähnlich funktionieren.
- Wie beim Sortieren existieren hier separate Lösungen, wenn z.B. die Schlüssel ganze Zahlen aus $\{0, \cdots, k-1\}$ sind

11 Graphen & Graphalgorithmen

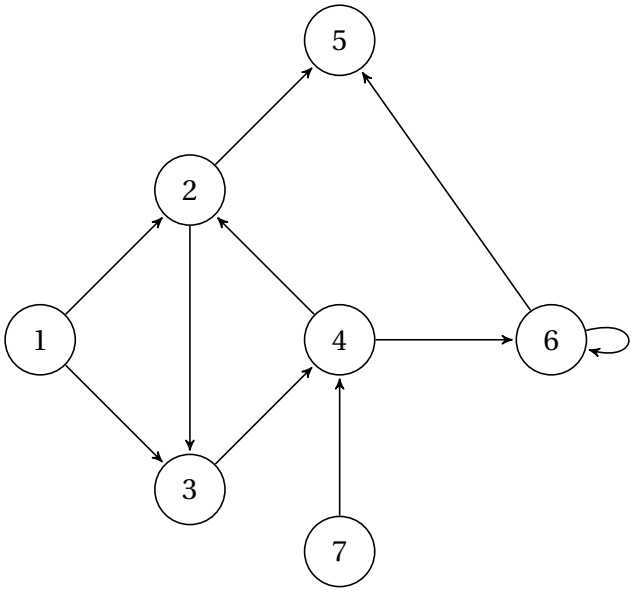
11.1 Definition

Ein gerichteter Graph $G = (V, E)$ besteht aus einer Menge V von **Knoten** und einer Menge $E \subseteq V \times V$ von **Kanten** (Vector/Edge). $e = (v, w) \in E$ heißt Kante von v nach w

11.2 Symbol



11.3 Beispiel



Die Kante $e = (v, w)$ ist inzident zum Knoten v (d.h. $source(e) = v$).w heißt dann Nachbarknoten von v oder adjazent.

Es gilt Für einen Knoten $v \in V$

$$\begin{aligned} outdeg(v) &= \#\text{ Aller zu v inzidenten Knoten} \\ &= |\{e \in E | v = source(e)\}| \\ &\text{heißt Ausgansgrad von } v \\ indeg(v) &= |\{e \in E | v = targe(e)\} \\ &\text{heißt Eingangsgrad von } v \end{aligned}$$

Im Beispiel

$$\begin{aligned} outdeg(1) &= 2 \\ indeg(1) &= 0 \end{aligned}$$

11.3.1 Beobachtung

$$\begin{aligned} \#\,\text{Kanten}(|E|) &= \sum_{v \in V} outdeg(v) \\ &= \sum_{v \in V} indeg(v) \end{aligned}$$

11.3.2 Bezeichnungen

$$n = |V|$$
$$m = |E|$$
$$m \leq n^2$$

VOLLSTÄNDIGER GRAPH $E = V \times V$ (alle Kanten vorhanden) $\Rightarrow\ m = n^2$

11.4 Pfad

Ein Pfad P ist eine Folge von Knoten $v_0 \cdots$ mit $(v_i, v_{i+i}) \in E$ für alle $i = 0, \cdots, l-1$ Pfad P von v nach $w \Leftrightarrow v_0 = v, v_l = w$

- l heißt Länge des Pfades
- P heißt Kreis, wenn $v_0 = v_l$
- P heißt einfach, wenn $v_i \neq v_j, für i \neq j$
- $\exists$ Pfad von v nach w, dann heißt w erreichbar
- leerer Pfad $l = 0$

SATZ Ein Graph $G = (V, E)$ ist zyklisch, falls G einen Kreis enthält (sonst azyklisch)

12 Datenstrukturen für gerichtete Graphen

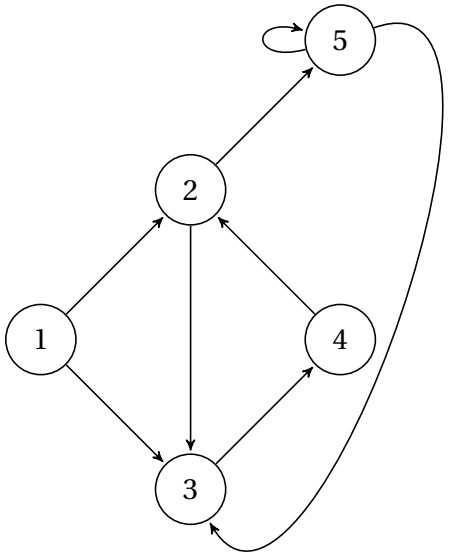
12.1 Möglichkeiten

12.1.1 Adjazenzmatrix (Nachbarschaftsmatrix)

Definition Boolsche $n \times m$-Matrix $A = (a_{i,j})_{1 \le i,j \le m} a_{i,j} = 1$, falls $(i,j) \in E$ 0, sonst

Beispiel

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	0	1
3	0	0	0	1	0
4	0	1	0	0	0
5	0	0	1	0	1



Vorteil Test, ob $(v,w) \in E$ in Zeit $\mathcal{O}(n)$

Nachteil Platzbedarf $\mathcal{O}(n^2)$ **!**

typische Operation

```
//Durchlaufe alle Nachbarn w von v
for i = 1 to n do
    if a_{v,w} = 1 then
        s. Übung
    end
end
```

Diese Iteration braucht hier immer Zeit $\mathcal{O}(n)$ besser $\mathcal{O}(outdeg(v))$

12.1.2 Adjazenzlisten

Speichere für jeden Knoten $v \in \{1, \cdots, n\}$ die Liste seiner Nachbarn, d.h. $\{w \in V | (v,w) \in E\}$ Feld $A[1, \cdots, n]$ von Listenköpfen. Der Test, ob $(v,w) \in E$ ist , ist hier teuer(wird allerdings (fast) nie gebraucht :D) Aber:

- Iteration über Nachbarn von v

```
foreach w ∈ V mit (v,w) ∈ E do
    Durchlaufe die Liste A() in Zeit O(outdeg(v)) = O(A[v].lenght)
end
```

- Platzbedarf $\mathcal{O}(n+m)$ (genauer:$n+\alpha m$ Speicherzellen) d.h. linear in der Größe des Graphen

```
class adj_elem{
int  node;
adj_elem  next;
}
```

12.2 topologische Sortierung

12.2.1 Definition

Eine topologische Sortierung eines Graphen $G = (V,E)$ mit $|v| = n$ ist eine Abbildung

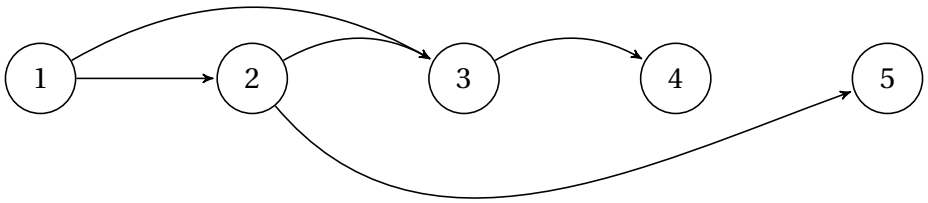
$$ord : V \to \{1, \cdots, n\}.$$

Es gilt zudem:

- ord ist injektiv
- $\forall (v,w) \in E : ord(v) < ord(w)$

Zusammenfassung Nummerierung der Knoten so, dass alle Kanten von kleineren zu größeren Nummern führen

INTUITIVE BESCHREIBUNG



BEOBACHTUNG Falls G zyklisch, existiert **keine** topologische Sortierung

12.2.2 ALGORITHMUS

```
count ← 0;
while G besitzt einen Knoten mit indeg = 0 do
    ord[v] ← ++count;
    G ← G\{v};
end
if G nicht leer then
    Error : „G zyklisch“
end
```

12.2.3 FOLGERUNGEN

SATZ Eine topologische Sortierung eines Graphen $G = (V, E)$ kann in Zeit $\mathcal{O}(n+m)$ berechnet werden, wobei $n = |V|$ und $m = |E|$

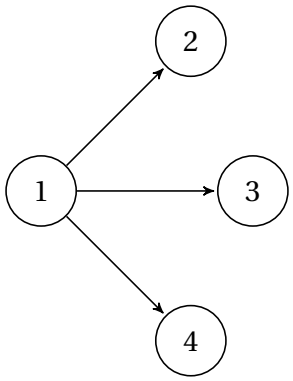
FOLGERUNG Test, ob G azyklisch hat auch Zeit $\mathcal{O}(n+m)$

12.3 SYSTEMATISCHE DURCHMUSTERUNG VON GRAPHEN

12.3.1 PROBLEM

Liste alle von einem Startknoten s aus erreichbaren Knoten (systematisch) auf

12.3.2 BEISPIEL

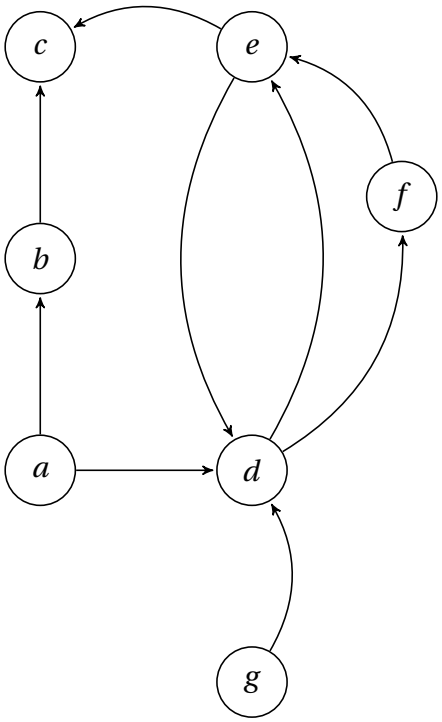


12.3.3 GRUNDLEGENDE STRATEGIEN

- Tiefensuche: Depth-First-Search (DFS)
- Breitensuch: Breadth-First-Search (BFS)

```
dfs_count ← 0;
comp_count ← 0;
foreach v ∈ V do
  | besucht[v] ← false;
end
[T, F, B, C] ← ∅;
foreach v ∈ V do
  | if ¬besucht[v] then
  |   | dfs(v);
  | end
end
```

12.3.3.1 BEISPIEL



KLASSIFIZIERUNG der Kanten mit Hilfe der Nummerierung d.h. dfsnum & compsum (und der Menge T)

LEMMA

- T, F, B, C ist Partition von E
- T entspricht dem Aufrufbaum der rekursiven Aufruf (→ DFS-Baum)

- $v \rightarrow w \Leftrightarrow dfsnum[v] \leq dfsnum[w] \wedge compnum,[v] \geq compnum[w]$
- $(v,w) \in T \cup F \Leftrightarrow dfsnum[v] < dfsnum[w]$
- $(v,w) \in B \Leftrightarrow dfsnum[v] \geq dfsnum[w] \wedge compnnum[v] \leq compnum[w]$
- $(v,w \in C \Leftrightarrow dfsnum[v] > dfsnum[w] \wedge compnum[v] > dfsnum[w]$

12.3.4 Folgerungen

- Partitionierung der Kanten T,F,B,C kann effizient berechnet werden in Zeit $\mathcal{O}(n+m)$
- In azyklischen Graphen produziert DFS keine Rückwärtskanten ($B = \emptyset$)

Lemma $\Rightarrow$ $\forall (v,w) \in E : compnum[v] > compnum[w]$
$\Rightarrow$ Die Abbildung $ord : V \rightarrow \{1,\cdots,n\}$ mit $ord(V) = n+1-compnum[v]$
Ist eine topologischer Sortierung des Graphen

12.3.5 Weitere Anwendungen

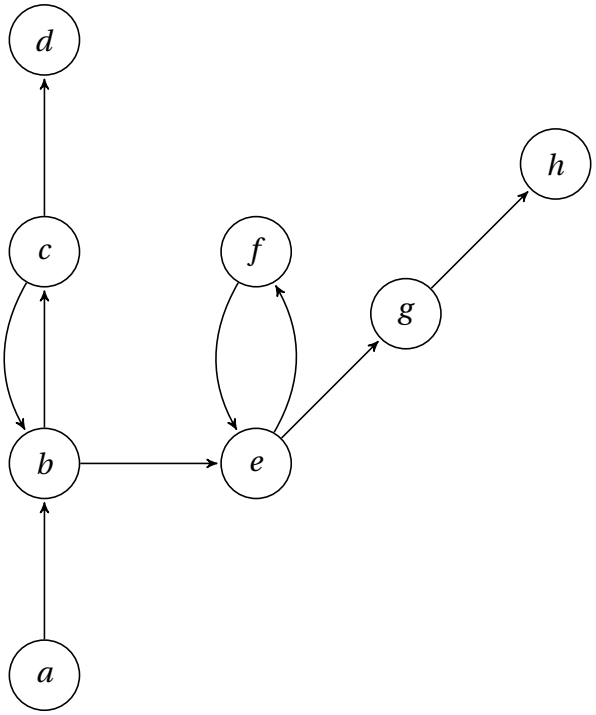
Berechnung der **starken Zusammenhangskomponenten** gerichteter Graphen

Definition

- Ein gerichteter Graph $G = (V,E)$ heißt stark zusammenhängenden, wenn $\forall v,w \in V : V \rightarrow W$ stark zusammenhängend
- Die starken Zusammenhangskomponenten(SZK) von G sind die maximal großen maximal starkzusammenhängenden Teilgraphen G.
- andere Darstellung
  Feld szknum mit Einträgen $\{1,2,3,4\}$

Idee für einen Algorithmus Führe DFS auf G aus! Sei $G' = (V',E')$ der Teilgraph der besuchten& benutzen Kanten. Verwalte die $SZKs$ von G'

STARTE SZK



DFS Verwalte die SZK von bereits besuchten Teilgraphen $G'(V', E')$

INITIALISIERUNG

$$V' = \{a\}$$
$$E' = \emptyset$$
$$SZK = \{\{a\}\}$$

Sei $(v, w) \in E$ die nächste von DFS betrachte Knoten $\Rightarrow \quad v \in V'$ (im Beispiel(a, b))

FALL 1 Füge eine neue Komponente $\{w\}$ zu der Menge ZSK hinzu
$ZSK \leftarrow SZK \cup \{\{w\}\}$

FALL 2 $(v, w) \notin T$ d.h. Vorwärts-,Rückwärts-,Crosskanten

12.3.6 KANTEN

VORWÄRTSKANTE Es passiert nichts, da keine neuen Pfade in G' entstehen

RÜCKWÄRTSKANTEN schließt einen Kreis $\Rightarrow$ eventuell mehrere Komponenten von G' zu einer einzigen Komponente vereinigt werden

CROSSKANTEN kann ebenfalls einen Kreis schließen

Dazu ein paar Definitionen:

- Eine SZK K heißt abgeschlossen, falls alle Aufrufe $dfs(v)$ für $v \in K$ abgeschlossen sind
- Die Wurzel v einer SZK K ist der Knoten mit der kleinsten $dfsnum$ in K
- unfertig bezeichnet eine Folge aller Knoten, für die dfs bereits aufgerufen wurde, aber deren SZK noch nicht abgeschlossen
- Wurzeln sind eine Unterfolge von Unfertigen, nach $dfsnum$ sortiert (nicht abgeschlossene SZK)

12.4 Beobachtungen

- Die Wurzel-Folge zerteilt die Unfertig-Folge in Intervalle, die alle nicht abgeschlossene $SZKs$
- $\forall$ Knoten v : $v \in$ unfertig $\Leftrightarrow v \rightarrow g$
- Wurzeln: Folge von Knoten auf aktuellen Baumpfad (Stack)
- $\nexists$ Kante (v,w) mit v in abgeschlossene und w in nicht abgeschlossene SZK

Nächster Schritt: Betrachte die Kanten aus g $(g,d) \in C$. Es passiert nichts, da d in abgeschlossene SZK
d.h. $d \notin$ unfertig $\Rightarrow$ kein Pfad von d nach g

12.5 Beobachtungen II

(g,d) schließt keinen Kreis, aber $(g,c) \in C$ schließt einen Kreis,da $C \in$ unfertig.
Die Vereinigung drei $SZKs$ mit Wurzeln b,e,g durch Löschen von e und g aus der Wurzelfolge.

Aktion Füge h hinten an Folgen unfertig und Wurzeln hinzu
Bei Rückkehr(Abschluss) eines Aufrufs $dfs(v)$ wird getestet, ob v eine Wurzel ist (v letztes Element der Wurzelliste).
Falls ja ist die SZK mit dieser Wurzel abgeschlossen
Dann wird v aus Wurzeln und die SZK aus unfertig entfernt.

12.5.1 Beachte

Hinzufügen und Entfernen von Knoten geschieht immer am rechten Ende $\Rightarrow$ Keller

```
while dfsnum[wurzel,top()]>dfsnum[w] do
  | wurzel.pop()
end
```

INDEX