Algorithmen und Datenstrukturen

Synix

Last update: August 8, 2023

Ein Modul des Informatikstudiums der TU Darmstadt

Disclaimer:
Diese LaTeX- Datei wurde lediglich aus Inhalten der Vorlesungen, deren Folien und Übungsblättern gebaut.
Ich übernehme keine Garantie für Richtigkeit und Vollständigkeit!
Garantie geben nur die offiziellen Materialien!

Contents

1 Introduction

1.1 Was sind Algorithmen und Datenstrukturen?

Zuerst schauen wir uns die Definition von Algorithmus und Datenstruktur an.

Algorithmen:
Definition: Eine aus endlich vielen Schritten bestehende, ausführbare Handlungsvorschrift zur eindeutigen Umwandlung von Eingabe- in Ausgabedaten.
Algorithmen haben folgende Characteristika:

- Anwendbar:
  - Allgemeinheit: Algorithmus für ganze Problemklasse anwendbar.
  - Korrektheit: Falls Algorithmus terminiert, ist die Ausgabe richtig.
- Berechenbar:
  - Finitheit: Algorithmus hat endliche Beschreibung.
  - Terminierung: Algorithmus stoppt in endlicher Zeit.
  - Effektivität: Schritte sind auf Maschine ausführbar.
- Bestimmt:
  - Determiniertheit: Algorithmus liefert bei gleicher Eingabe gleiche Ausgabe.
  - Determinismus: Algorithmus durchläuft für gleiche Eingabe immer die gleichen Schritte/Zustände.

Datenstrukturen:
Definition: Eine Datenstruktur ist eine Methode, Daten für den Zugriff und die Modifikation zu organisieren.
Datenstrukturen stellen sich aus Daten zusammen, die in einer Struktur gespeichert sind, als Besipiel ein Array in Java:

```
1 int[] integerArray = {1, 4, 2, 8, 6};
```

oder wir initialisieren den Array zuerst und fügen die Werte danach hinzu:

```
1 int[] integerArray = new int[5];
2 for(int i = 0; i< integerArray.length; i++){
3     integerArray[i] = i;
4 }
```

Später werden wir uns hiermit genauer beschäftigen.

2 Sorting

2.1 Sortierproblem

Es gibt eine Folge von Objekten und ziel ist es, diese zu sortieren (gemäß eines Schlüsselwerts). Die zu sortierenden Daten werden als "Satellitendaten" bezeichnet.

Schlüsselproblem
Schlüsselwerte müssen nicht eindeutig, aber müssen "sortierbar" sein. Wir nehmen an, es gibt eine totale Ordnung $\leq$ auf der Menge M aller möglichen Schlüsselwerte.

Totale Ordnung
Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M:
Die Relation $\leq$ auf M ist genau dann eine totale Ordnung, wenn gilt.

- Reflexivität: $\forall x \in M : x \leq x$
- Transitivität: $\forall x, y, z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Antisymmetrie: $\forall x, y \in M : x \leq y \wedge y \leq x \Rightarrow x = y$
- Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$

2.2 Insertion- Sort

Die Idee von Insertion- Sort ist, dass wir einen Array von Zahlen, bzw. vergleichbaren Elementen, durchgehen und diese durch herrausnehmen und wiedereinfügen sortieren. Hier ein Python Programm:

```
1 insertionSort(A)
2     FOR i=1 TO A.length-1 DO
3         // insert A[i] in pre-sorted sequence A[0..i-1]
4         key=A[i];
5         j=i-1; // search for insertion point backwards
6         WHILE j>=0 AND A[j]>key DO
7             A[j+1]=A[j]; // move elements to right
8             j=j-1;
9         A[j+1]=key;
```

Wir gehen hier von links nach rechts den Array durch, schauen und die Zahl an und vergleichen ihn mit allen Linken. Sobald die Zahl, die er gerade betrachtet, kleiner ist als der Key, wird die Zahl bei j+1 eingefügt, andernfalls wird array[j] auf array[j+1] verschoben.

2.3 Laufzeitanalyse

Zentrale Frage: Wieviel Schritte macht Algorithmus in Abhängigkeit von der Eingabekomplexität?
Analysiere, wie oft jede Zeile maximal ausgeführt wird.

Jede Zeile i hat Aufwand ci.

n Anzahl zu sortierender Elemente

Zeile	Aufwand	Anzahl
1	c1	n
2	c2	$n-1$
3	c3	$n-1$
4	c4	
5	c5	
6	c6	
7	c7	$n-1$

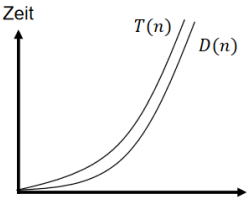
Figure 1: Beispiel von Insertion Sort

```
1 insertionSort(A)
2     FOR i=1 TO A.length-1 DO //insert A[i] in pre-sorted
          sequence A[0..i-1]
3         key=A[i];
4         j=i-1; // search for insertion point backwards
5         WHILE j>=0 AND A[j]>key DO
6             A[j+1]=A[j]; // move elements to right
7             j=j-1;
8         A[j+1]=key;
```

Bei Insertion- Sort bedeutet das, dass Zeilen 4, 5 und 6 im schlimmsten Fall bis $j = -1$ also jeweils i-mal ausgeführt werden. Insgesamt: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (Zeile 4 jeweils einmal mehr bis Abbruch)
Maximale Gesamtlaufzeit Insertion-Sort: $T(n) = c2 \cdot n + (c3 + c4 + c5 + c8) \cdot (n-1) + (c5 + c6 + c7) \cdot \frac{n(n-1)}{2}$. Wie teuer ist z.B. Zuweisung $A[j+1] = A[j]$ in Zeile 5, also was ist $c6$? Hängt stark von Berechnungsmodell ab (in dem Pseudocode-Algorithmus umgesetzt wird)... . Wnehmen üblicherweise an, dass elementare Operationen (Zuweisung, Vergleich,...) in einem Schritt möglich $\rightarrow c6 = 1$.

Gesamtlaufzeit Insertion-Sort (mit $ci = 1$):
$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2}$

Zum Vergleich (nur dominanter Term) :
$D(n) = 3 \cdot \frac{n(n-1)}{2}$



n	$T(n)$	$D(n)$	relativer Fehler $(T(n) - D(n))/T(n)$
100	15.346	14.850	3,2321 %
1.000	1.503.496	1.498.500	0,3323 %
10.000	150.034.996	149.985.000	0,0333 %
100.000	15.000.349.996	14.999.850.000	0,0033 %
1.000.000	150.000.034.999.996	1.499.998.500.000	0,0003 %

Figure 2: Asymptotische Vereinfachung

Weiter Vereinfachung (nur abhängiger Term): $A(n) = \frac{n(n-1)}{2}$.

2.3.1 Θ- Notation

Auch Landau- Notation genannt.
Wie nehmen eine Funktion $f, g : N \to R_{>0}$.

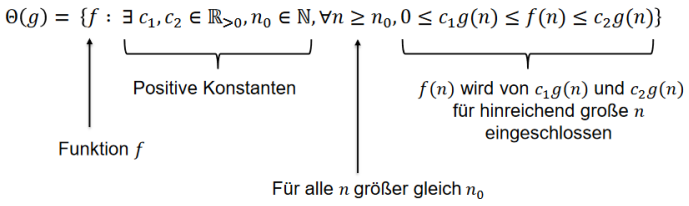


Figure 3: Definition

Schreibweise: $f \in \Theta(g)$, manchmal auch $f = \Theta(g)$.

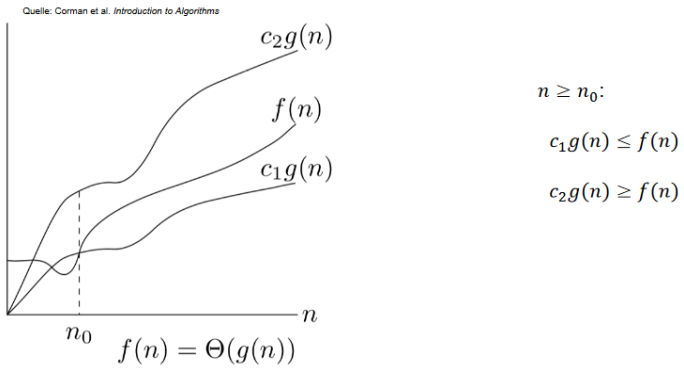


Figure 4: Veranschaulichung Θ-Notation

$g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$. Θ-Notation beschränkt eine Funktion asymptotisch von oben und unten.
Bei Insertion Sort ergibt sich mit der oben angegebenen Formel $T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2} = \Theta(n^2)$:

- Untere Schranke ($c_1 = \frac{3}{2}$): $\frac{3}{2}n^2$ für $n \geq 2$
- Obere Schranke ($c_2 = 7$): $7 \cdot n^2$

2.3.2 0- Notation

Auch als *Obere Asymptotische Schranke* bezeichnet.

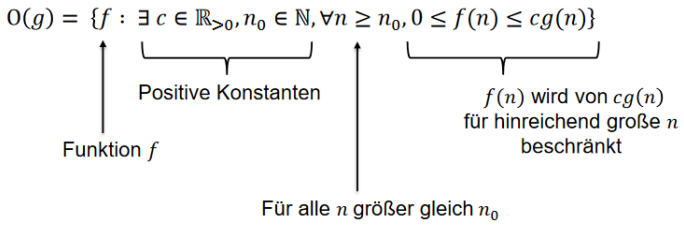


Figure 5: Definition

Sprechweise: f wächst höchstens so schnell wie g.
Schreibweise: $f = O(g)$ oder auch $f \in O(g)$.

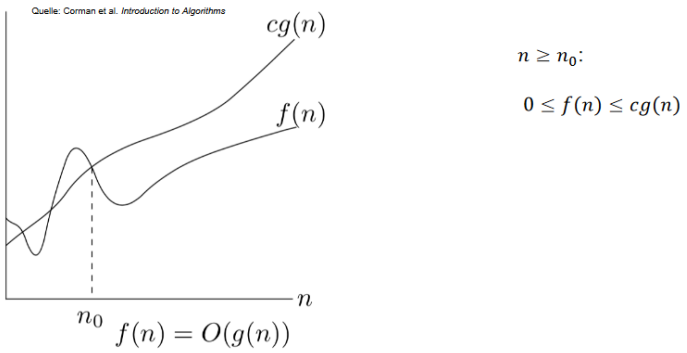


Figure 6: Veranschaulichung O-Notation

Beachte: $\Theta(g(n)) \subseteq O(g(n))$ und somit $f(n) = \Theta(g) \Rightarrow f(n) = O(g)$.
Rechenregeln:

- Konstanten: $f(n) = a$ mit $a \in R_{>0}$ konstant. Dann $f(n) = O(1)$
- Skalare Multiplikation: $f = O(g)$ und $a \in R_{>0}$. Dann $a \cdot f = O(g)$
- Addition: $f_1 = O(g_1)$ und $f_2 = O(g_2)$. Dann ist $f_1 + f_2 = O(max\{g_1, g_2\})$
- Multiplikation: $f_1 = O(g_1)$ und $f_2 = O(g_2)$. Dann $f_1 \cdot f_2 = O(g_1 \cdot g_2)$

2.3.3 Ω- Notation

Auch als *Untere Asymptotische Schranke* bezeichnet.

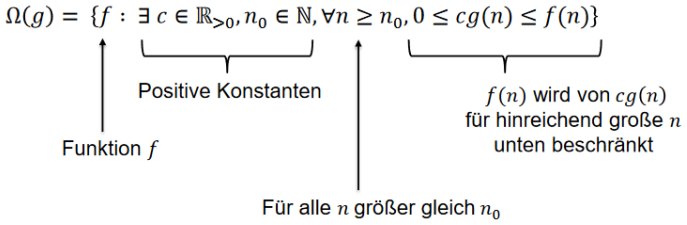


Figure 7: Definition

Sprechweise: f wächst mindestens so schnell wie g.
Schreibweise: $f = \Omega(g)$ oder auch $f \in \Omega(g)$.

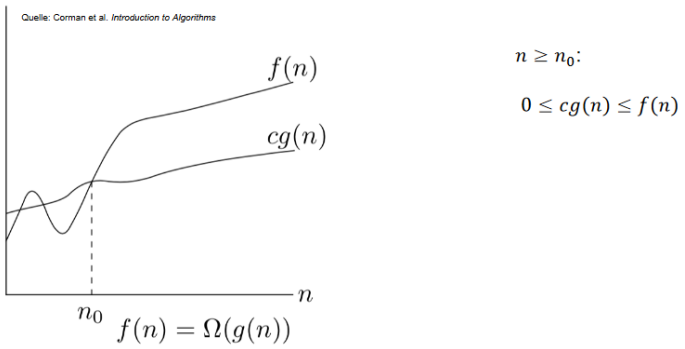


Figure 8: Veranschaulichung Ω-Notation

Beachte: $\Theta(g(n)) \subseteq \Omega(g(n))$ und somit $f(n) = \Theta(g) \Rightarrow f(n) = \Omega(g)$.

2.3.4 Komplexitätsklassen

- $\Theta(1)$: Konstant (Bsp: Einzeloperation)
- $\Theta(logn)$: Logarithmisch (Binäre Suche)
- $\Theta(n)$: Linear (Sequentielle Suche)
- $\Theta(nlogn)$: Quasilinear (Sortieren eines Arrays)
- $\Theta(n^2)$: Quadratisch (Matrixaddition)
- $\Theta(n^3)$: Kubisch (Matrixmultiplikation)
- $\Theta(n^k)$: Polynomiell
- $\Theta(2^n)$: Exponentiell (Travelling-Salesman*)
- $\Theta(n!)$: Faktoriell (Permutationen)

2.3.5 o-Notation und ω-Notation

$o(g) = \{f : \forall c \in R_{>0}, \exists n_0 \in N, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$ (Beispiel: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$).
$\omega(g) = \{f : \forall c \in R_{>0}, \exists n_0 \in N, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$ (Beispiel: $\frac{n^2}{2} = \omega(n)$ und $\frac{n^2}{2} \neq \omega(n^2)$).

2.4 Merge- Sort

Auch *Divide and Conquer- Algorithmus* genannt.

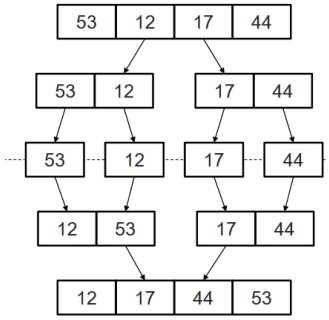


Figure 9: Visualisierung des Merge Sorts

Teile Liste in Hälften, sortiere (rekursiv) Hälften, sortiere wieder zusammen.

```
mergeSort(A,l,r) // initial call: l=0,r=A.length-1
    IF l<r THEN //more than one element
        m=floor((l+r)/2); // m (rounded down) middle (
            genauer: letzter Index des linken Teils)
        mergeSort(A,l,m); // sort left part
        mergeSort(A,m+1,r); // sort right part
        merge(A,l,m,r); // merge into one
```

Wir sortieren im Array A zwischen Position l (links) und r (rechts).
Merge:
Idee : nimm nächstes kleinstes Element aus linker oder rechter Teilliste und gehe in dieser Liste eine Position nach rechts.
Jede Liste wird genau einmal durchlaufen $\Rightarrow$ Laufzeit $\Theta(n_L + n_R)$.

```
merge(A,l,m,r) // requires l<=m<=r; array B with r-l+1
    elements as temporary storage
    pl=l; pr=m+1; // position left, right
    FOR i=0 TO r-l DO // merge all elements
        IF pr>r OR (pl=<m AND A[pl]=<A[pr]) THEN
            B[i]=A[pl];
            pl=pl+1;
        ELSE //next element at pr
            B[i]=A[pr];
            pr=pr+1;
    FOR i=0 TO r-l DO A[i+l]=B[i]; //copy back to A
```

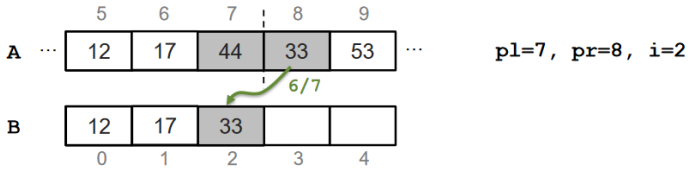


Figure 10: Beispiel Merge, wenn er in einem Array dargestellt wird

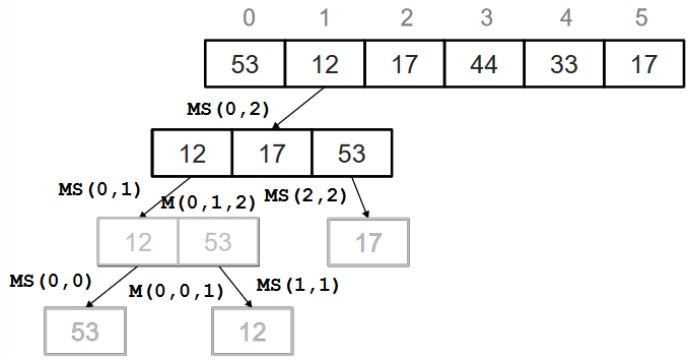


Figure 11: Beispiel Merge- Sort aus der Mitte des Algorithmus

2.5 Mastertheorem

Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nicht-negativen ganzen Zahlen durch die Rekursionsgleichung $T(n) = aT(\frac{n}{b}) + +f(n)$, $T(1) = \Theta(1)$ definiert, wobei wir $\frac{n}{b}$ so interpretieren, dass damit entweder $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

- 1. Gilt $f(n) = O(n^{log_b(a)-\gamma})$ für eine Konstante $\gamma > 0$, dann $T(n) = \Theta(n^{log_b a})$
- 2. Gilt $f(n) = \Theta(n^{log_b a})$, dann gilt $T(n) = \Theta(n^{log_b a} \cdot log_2 n)$
- 3. Gilt $f(n) = \Omega(n^{log_b(a)+\gamma})$ für eine Konstante $\gamma > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Interpretation (entscheident ist Verhältnis von $f(n)$ zu $n^{log_b a}$)

- 1. Wenn $f(n)$ polynomiell kleiner als $n^{log_b a}$, dann $T(n) = \Theta(n^{log_b a})$
- 2. Wenn $f(n)$ und $n^{log_b a}$ gleiche Größenordnung, dann $T(n) = \Theta(n^{log_b a} \cdot logn)$

- 3. Wenn $f(n)$ polynomiell größer als $n^{log_b a}$ und $af(n/b) \leq cf(n)$, dann $T(n) = \Theta(f(n))$

'Regularität' $af(n/b) \leq cf(n)$, $c < 1$ in Fall 3: $T(n) = a \cdot T(n/b) + f(n) = a \cdot (a \cdot T(n/b^2) + f(n/b)) + f(n)$.
Aufwand $f(n)$ zum Teilen und Zusammenfügen für Größe n dominiert (asymptotisch) Summe $af(n/b)$ aller Aufwände für Größe n/b braucht man nur im dritten Fall für 'große' $f(n)$
Der Unterschied zwischen 1 und 3 ist der polynomielle Faktor n^{γ}.

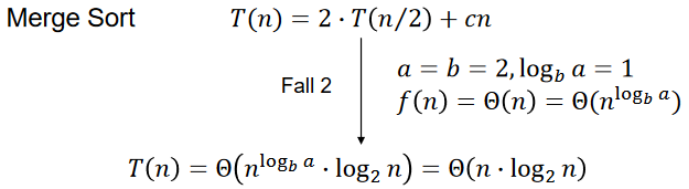


Figure 12: Beispiel Mastertheorem von Merge- Sort

2.6 Quick- Sort

Wie in Merge- Sort verwendet Quicksort Divide and Conquer- Ansatz.
Quicksort steckt mehr Arbeit in Aufteilen, Zusammenfügen kostenlos.

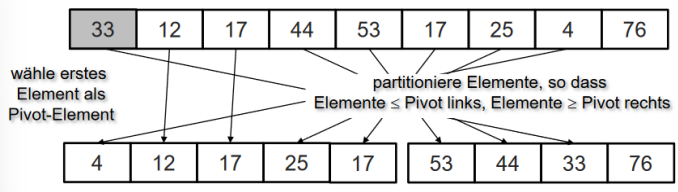


Figure 13: Erläuterung

Sortiert beide Teil-Arrays rekursiv, wobei das Ergebnis ein komplett sortiertes Array ist.

```
quicksort(A,l,r) // initial call: l=0,r=A.length-1
    IF l<r THEN //more than one element
        p=partition(A,l,r); // p partition index
        quicksort(A,l,p); // sort left part
        quicksort(A,p+1,r); // sort right part
```

```
partition(A,l,r) //requires l<r, returns int in l..r-1
    pivot=A[l];
    pl=l-1; pr=r+1; //move from left resp. right
    WHILE pl<pr DO
        REPEAT pl=pl+1 UNTIL A[pl]>=pivot; //move left up
        REPEAT pr=pr-1 UNTIL A[pr]=<pivot; //move right down
        IF pl<pr THEN Swap(A[pl],A[pr]);
        p=pr; //store current value
    RETURN p // A[l..p] left, A[p+1..r] right
```

Hier ein Beispiel von Partition, mit *pivot=12, l=2, r=6*:

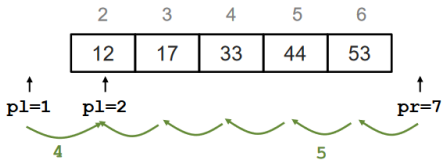


Figure 14: Beispiel der Suche nach den zu tauschenden Elementen

Daraus folgt P*p=2*.

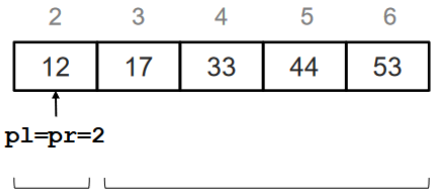


Figure 15: Beispiel bei der Findung des Elements

In diesem Fall haben wir jedoch pl=pr, deswegen kein Swap.
Ein Beispiel- Swap wäre:

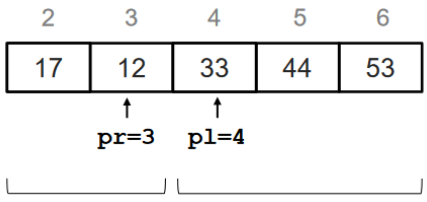


Figure 16:

Daraufhin werden 12 und 17 geswaped (mit *pivot=33, p=3*).

Laufzeit Partition:
Für jede Erhöhung von pl bzw. Erniedrigung von pr konstant viele Schritte:

- pl und pr haben zu Beginn Abstand $n+2$ und bewegen sich in jeder Iteration aufeinander zu $\Rightarrow O(n)$
- pl und pr bewegen sich in jeder einzelnen REPEAT-Iteration maximal 1 aufeinander zu $\Rightarrow \Omega(n)$

Laufzeit Partition: $\Theta(n)$.

2.6.1 Laufzeitanalyse QuickSort

Worst-Case:

- **Untere Schranke:** Ungünstiges Array: Partition spaltet immer nur ein Element ab. $(n-1)$-mal Partition ergibt Quicksort Gesamtlaufzeit$\Omega(n^2)$.
- **Obere Schranke:** $T(n) \leq dn^2$

Quicksort (Worst-Case-)Laufzeit: $\Theta(n^2)$
Best- Case:

- Im besten Fall Aufteilung in gleichgroße Arrays wie bei Merge Sort: $T(n) = 2T(n+2) + \Theta(n) \Rightarrow$ Laufzeit $\Theta(n \cdot logn)$
- Gilt auch, solange beide Arrays in Größenordnung $\Omega(n)$, z.B. stets 10% der n Elemente links und 90% rechts: $T(n) = T(0.1n) + T(0.9n) + \Theta(n)$ Laufzeit $\Theta(n \cdot logn)$

Average:
Wie verhält sich Quicksort im Durchschnitt auf 'zufälliger' Eingabe?
Für zufällige Permutation $D(n)$ eines fixen Arrays von n Elementen benötigt Quicksort $E_{D(n)}[AnzahlSchritte] = O(n \cdot logn)$

Insertion Sort	Merge Sort	Quicksort
Laufzeit $\Theta(n^2)$	Beste asymptotische Laufzeit $\Theta(n \log n)$	Wort-Case-Laufzeit $\Theta(n^2)$, randomisiert mit erwarteter Laufzeit $\Theta(n \log n)$
einfache Implementierung		in Praxis meist schneller als Merge Sort, da weniger Kopieroperationen
für kleine $n \leq 50$ oft beste Wahl		Implementierungen schalten für kleine n meist auf Insertion Sort

Figure 17: Vergleich: Insertion, Merge, und Quicksort

2.7 Radix- Sort

Ansatz: Schlüssel sind d-stellige Werte in D-närem Zahlensystem:

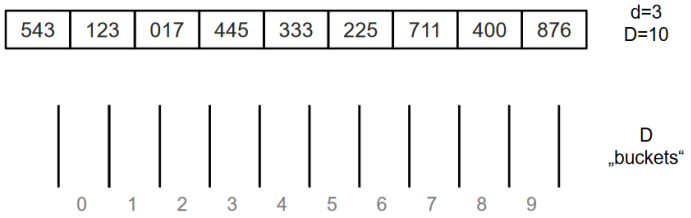


Figure 18: Beispiel

'Buckets' erlauben Einfügen und dann Entnehmen (in eingefügter Reihenfolge) in konstanter Zeit.
Dabei gehen wir alle Zahlen von der kleinstwertigen zur größtwertigsten Stelle durch und fügen diese in die Buckets ein, danach verschieben wir diese wieder in den Array.

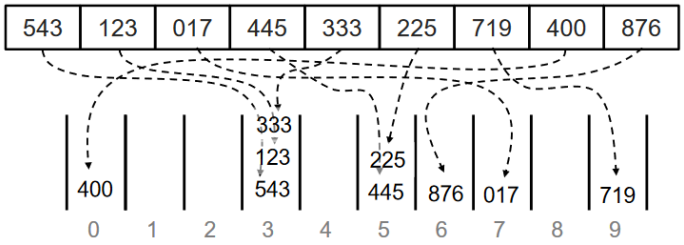


Figure 19: Erste Iteration des Beispiels, also kleinstwertige Stelle

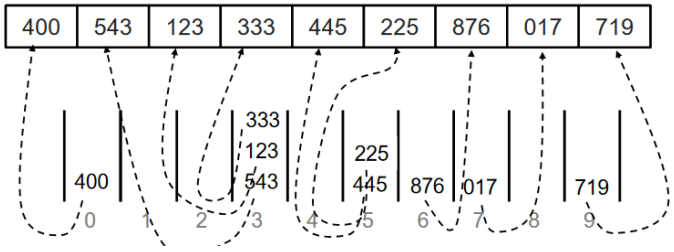


Figure 20: Einfügen nach dem aufteilen in die Buckets

Nachdem wir dies für jede Ziffer machen, haben wir eine sortierte Liste.

```
radixSort(A) // keys: d digits in range [0,D-1]; B[0][],...,
    B[D-1][] buckets (init: B[k].size=0)
    FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
        FOR j=0 TO n-1 DO putBucket(A,B,i,j);
        a=0;
        FOR k=0 TO D-1 DO //rewrite to array
            FOR b=0 TO B[k].size-1 DO
                A[a]=B[k][b]; //read out bucket in order
                a=a+1;
            B[k].size=0; //clear bucket again
    RETURN A;
```

Laufzeit: $O(d \cdot (n + D))$.

```
putBucket(A,B,i,j) // call-by-reference
    z=A[j].digit[i]; // i-th digit of A[j]
    b=B[z].size; // next free spot
    B[z][b]=A[j];
    B[z].size=B[z].size+1;
```

laufzeit: $O(1)$.

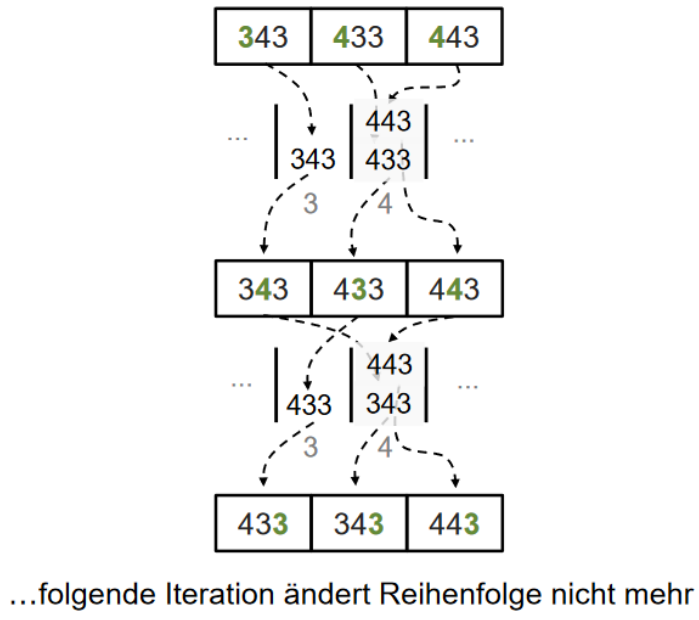


Figure 21: Warum man nicht mit der größtwertigsten Ziffer beginnen kann

3 Basic Data Strctures

3.1 Stacks

Als Stacks bezeichnet man eine abstrakte Datenstruktur, die mit den LIFO-Prinzip arbeitet (Last in, First out). Dabei haben Stacks folgende Operationen:

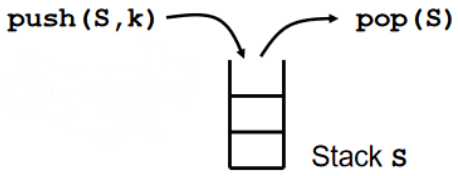


Figure 22: Visualisierung von LIFO eines Stacks S

- new(S) - erzeugt neuen (leeren) Stack namens S
- isEmpty(S) - gibt an, ob Stack S leer
- pop(S) - gibt oberstes Element vom Stack S zurück und löscht es vom Stack (bzw. Fehlermeldung, wenn Stack leer)
- push(S,k) - schreibt k als neues oberstes Element auf Stack S (bzw. Fehlermeldung, wenn Stack voll)

Man kann einen Stack auch als Array darstellen. Dabei wird das Element im Array, welches den größten Index hat, als Top gesehen.
Wenn man den Stack als Array darstellt, ergeben sich folgende Methoden:

```
new(S)
    S.A[]=ALLOCATE(MAX);
    S.top=-1;
```

```
isEmpty(S)
    IF S.top<0 THEN
        RETURN true;
    ELSE
        RETURN false;
```

```
pop(S)
    IF isEmpty(S) THEN
        error "underflow";
    ELSE
        S.top=S.top-1;
        RETURN S.A[S.top+1];
```

Laufzeit: $\Theta(1)$

```
push(S,k)
    IF S.top==MAX-1 THEN
        error "overflow";
    ELSE
        S.top=S.top+1;
        S.A[S.top]=k;
```

Laufzeit: $\Theta(1)$
Man kann auch Stacks mit Variabler Größe machen, bei der bei vollem Stack immer ein neuer Stack erstellt wird (Effizient: Größe x2) und alle bisherige Einträge kopiert werden.
Laufzeitanalyse, gilt für Vergrößerung und Verkleinerung:
(1) n Elemente (unmittelbar nach letzter Vergrößerung)
(2) neue Speichergrenze wird nur erreicht, wenn dann mindestens n viele Push-Befehle
(3) Umkopieren kostet dann $O(n)$ Schritte
Im Durchschnitt für jeden der mindestens n Befehle $\Theta(n)$ Umkopierschritte!

3.2 Verkettete Listen

Eine verkettete Liste ist eine Datenstruktur, bei der das Element immer zusätzlich auf den Vorgänger und den Nachfolger verweist.

- key: Wert
- prev: Zeiger auf Vorgänger (oder *nil*)
- next: Zeiger auf Nachfolger (oder *nil*)

Zusätzlich noch head (Zeigt auf das erste Element (*nil* bei leerer Liste)).
Des weiteren gibt es noch folgende Elementare Operationen:

```
search(L,k) //RETURNs pointer to k in L (or nil)
    current=L.head;
    WHILE current != nil AND current.key != k DO
        current=current.next;
    RETURN current;
```

Laufzeit: $\Theta(n)$

```
insert(L,x) //inserts element x in L
    x.next=L.head;
    x.prev=nil;
    IF L.head != nil THEN
        head.prev=x;
    L.head=x;
```

Laufzeit: $\Theta(1)$
Hierbei wird nicht überprüft, ob das Element vorhanden ist. Wenn man vorher noch sucht, dann hat man Laufzeit $\Omega(n)$.

```
delete(L,x) //deletes element x from L
    IF x.prev != nil THEN
        x.prev.next=x.next
    ELSE
        L.head=x.next;
    IF x.next != nil THEN
        x.next.prev=x.prev;
```

Laufzeit: $\Theta(1)$
Es gibt auch noch einen Sentinel, welcher ein nicht sichtbarer Knoten ist, für den gilt (L ist die Liste):

- L.sent = head.prev
- head = L.sent.next
- Leere Liste besteht nur aus Sentinel

Eine angepasste Operation wäre:

```
deleteSent(L,x) // deletes x from L with sentinel
    x.prev.next=x.next;
    x.next.prev=x.prev;
```

Andere Operationen müssen auch angepasst werden.

3.3 Queues

Als Queue bezeichnet man eine abstrakte Datenstruktur, die mit den FIFO-Prinzip arbeitet (First in, First out). Dabei hat eine Queue folgende Operationen:

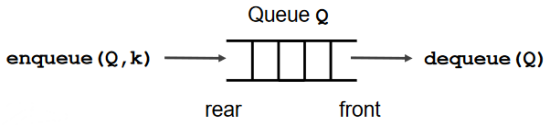


Figure 23: Visualisierung von FIFO einer Queue Q

- new(Q) - erzeugt neue (leere) Queue namens Q
- isEmpty(Q) - gibt an, ob Queue Q leer
- dequeue(Q) - gibt vorderstes Element der Queue Q zurück und löscht es aus Queue (bzw. Fehlermeldung, wenn Queue leer)
- enqueue(Q,k) - schreibt k als neues hinterstes Element auf Q (bzw. Fehlermeldung, wenn Queue voll)

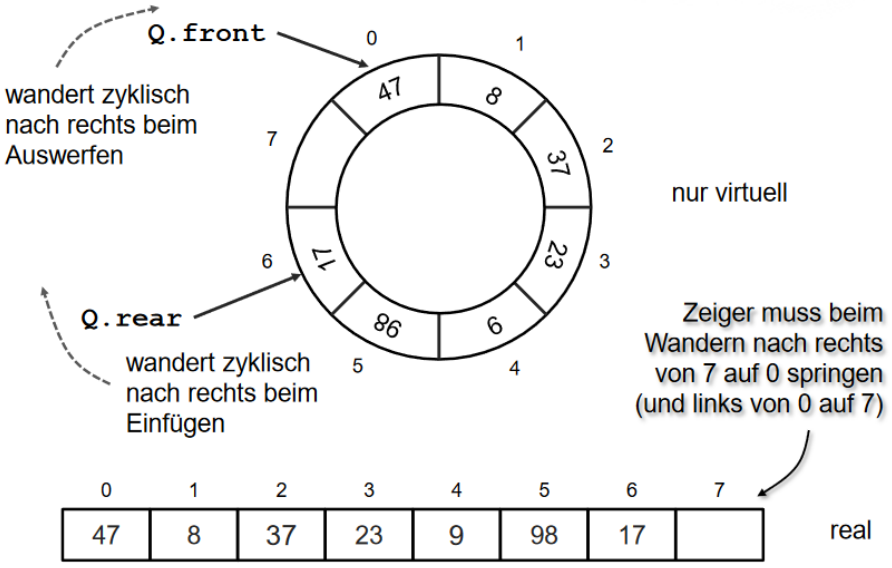


Figure 24: Queues als (virtuelles) zyklisches Array

Man kann auch Queues als Array darstellen, jedoch ist dies schwierig. Denn entweder ist der Array irgendwann voll, oder es wird Speicherplatz verschwendet. Eine Lösung ist ein zyklischer Array: Um zu Wissen, ob die zyklische Queue voll oder leer ist, muss man dies in einer Variable speichern, oder alternativ ein Element des Arrays als Abstandhalter reservieren. Algorithmen bei Queues als zyklischer Array:
Hierbei gilt: Q leer, wenn front == rear und empty == true und voll, wenn front == rear und empty = false

```
new(Q)
    Q.A[]=ALLOCATE(MAX);
    Q.front=0;
    Q.rear=0;
    Q.empty=true;
```

```
isEmpty(Q)
    RETURN Q.empty;
```

```
dequeue(Q)
    IF isEmpty(Q) THEN
        error "underflow";
    ELSE
        Q.front=Q.front+1 mod MAX;
        IF Q.front==Q.rear THEN
```

```
7             Q.empty=true;
8         RETURN Q.A[Q.front-1 mod MAX];
```

Laufzeit: $\Theta(1)$

```
1  enqueue(Q,k)
2  IF Q.rear==Q.front AND !Q.empty
3      error "overflow";
4  ELSE
5      Q.A[Q.rear]=k;
6      Q.rear=Q.rear+1 mod MAX;
7      Q.empty=false;
```

Laufzeit: $\Theta(1)$
Man kann eine Queue auch durch eine einfache! verkettete Liste darstellen, wessen Algorithmen aber leichter sind.

3.4 Binärer Baum

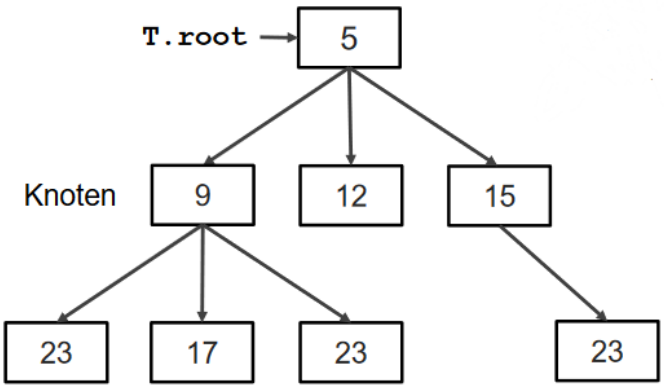


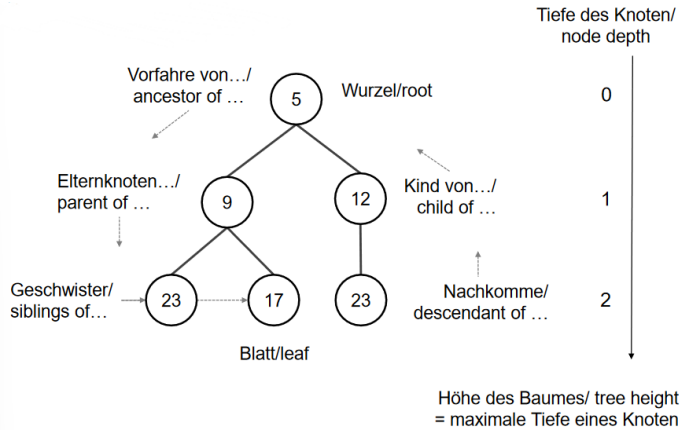
Figure 25: Darstellung eines binären Baums

Das ist ein binärer Baum.
Jeder Knoten enthält:

- key - Wert
- child[] - Array von Zeigern auf Kinder
- parent - Zeiger auf Elterknoten (oder bei root *nil*)

Baum-Bedingungen: Baum ist leer oder es gibt einen Knoten r (Wurzel), so dass jeder Knoten v von der Wurzel aus per eindeutiger Sequenz von child-Zeigern erreichbar ist: v = r.child[i1].child[i2]. . . . .child[im].
Außerdem sind Bäume azyklisch (Man kann keinen Zyklus bilden).

Man kann Binäre Bäume auch als ungerichtete Graphen darstellen, welches wir ab jetzt tun werden.
Begrifflichkeiten:



Außerdem gilt:

- Jeder Knoten hat maximal zwei Kinder: left=child[0] und0 right=child[1]
- Ein "Halbblatt" hat genau ein Kinder
- Ein Knoten z hat einen linken und rechten Teilbaum, falls left und right != nil
- Als Teilbaum bezeichnet man alle Knoten, die sich auf dieser Seite des Knotens befinden (Bsp: child[0], child[0].child[0], child[0].child[1]...)
- Höhe (Nicht leerer) Baum = maxHöhe aller Teilbäume der Wurzel +1
- Höhe leerer Baum hier per Konvention -1

Traversierungen von Binärbäumen:
Nehmen wir für die Beispiele die Abbildung der Begrifflichkeiten.
Inorder- Traviersierung:

```
1 inorder(x)
2     IF x != nil THEN
3         inorder(x.left);
4         print x.key;
5         inorder(x.right);
```

Beispiel: inorder(T.root) = [23 , 9 , 17 , 5 , 12 , 23]
Wichtig ist noch: Verschiedene Bäume können gleiche Inorder haben.
Preorder- Traviersierung:

```
preorder(x)
    IF x != nil THEN
        print x.key;
        preorder(x.left)
        preorder(x.right);
```

Beispiel: preorder(T.root) = [5 , 9 , 23 , 17 , 12 , 23]
Wichtig ist noch: Verschiedene Bäume können gleiche Preorder haben.
Postorder- Traviersierung:

```
postorder(x)
    IF x != nil THEN
        postorder(x.left)
        postorder(x.right);
        print x.key;
```

Beispiel: postorder(T.root) = [23 , 17 , 9 , 23 , 12 , 5]
Dabei ergeben jedoch Preorder + Inorder + eindeutige Werte einen Binärbaum:

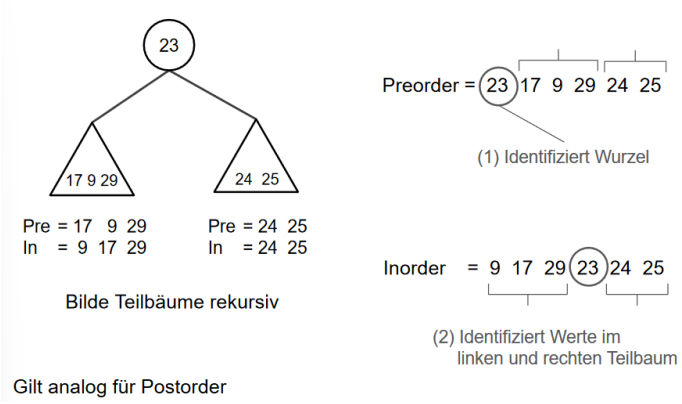


Figure 26: Beispiel

3.4.1 Abstrakter Datentyp- Baum

Operationen:

```
search(x,k)
    IF x==nil
        RETURN nil;
    IF x.key==k
        RETURN x;
    y=search(x.left,k);
```

```
7       IF y != nil
8           RETURN y;
9       RETURN search(x.right,k);
```

Dabei startet man mit search(T.root, k); **Laufzeit:** $\Theta(n)$

```
1   insert(T,x)
2       IF T.root != nil THEN
3           T.root.parent=x;
4           x.left=T.root;
5       T.root=x;
```

Laufzeit: $\Theta(1)$
Wenn der eingefüge Knoten die neue Wurzel ist, ensteht ein einsietiger Baum.

```
1   delete(T,x) //assumes x in T
2       y=T.root;
3       WHILE y.right!=nil DO
4           y=y.right;
5       connect(T,y,y.left);
6       IF x != y THEN
7           y.left=x.left;
8           IF x.left != nil THEN
9               x.left.parent=y;
10          y.right=x.right;
11          IF x.right != nil THEN
12              x.right.parent=y;
13      connect(T,x,y);
```

Beim löschen muss man beachten, dass wenn ein Halbblatt gelöscht wird, der Knoten durch das Halbblatt ganz rechts ersetzt wird, aber es geht auch ander:

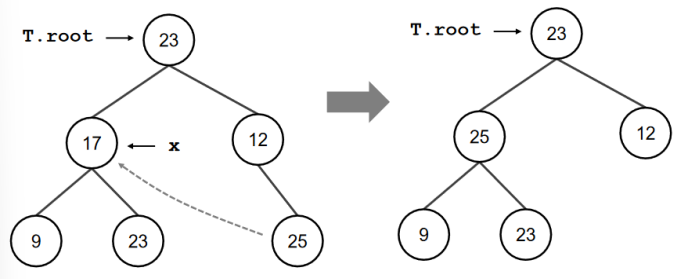


Figure 27: Visualisierung

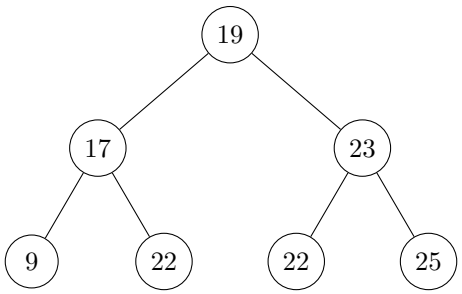
Laufzeit: $\Theta(h)$ Allgemein ist das Löschen aber je nach Knoten sehr individuell.

3.5 Binärer Suchbaum (BST)

Ein binärer Suchbaum ist ein Binärer Baum, bei dem wir wieder totale Ordnung annehmen.
Im Binären Suchbaum gilt für jeden Knoten z:

- Wenn x Knoten im linken Teilbaum von z, dann $x.key <= z.key$
- Wenn y Knoten im rechten Teilbaum von z, dann $y.key >= z.key$

Beispiel:



Zumal gilt:

- preorder + eindeutige Werte = Binärer Suchbaum
- inorder + eindeutige Werte $\neq$ Binärer Suchbaum

```
search(x,k) //1.Aufruf x=root
    IF x==nil OR x.key==k THEN
        RETURN x;
    IF x.key > k THEN
        RETURN search(x.left,k)
    ELSE
        RETURN search(x.right,k);
```

Laufzeit: $O(h)$, h 0 Höhe des Baumes

```
iterative-search(x,k) //Aufruf x=root
    WHILE x != nil AND x.key != k DO
        IF x.key > k THEN
            x=x.left
        ELSE
            x=x.right;
    RETURN x;
```

```
insert(T,z) //may insert z again z.left==z.right==nil;
    x=T.root; px=nil;
    WHILE x != nil DO
        px=x;
```

```
        IF x.key > z.key THEN
            x=x.left
        ELSE
            x=x.right;
    z.parent=px;
    IF px==nil THEN
        T.root=z
    ELSE
        IF px.key > z.key THEN
            px.left=z
        ELSE
            px.right=z;
```

Laufzeit: $O(h)$

Löschen:

```
delete(T,z)
    IF z.left==nil THEN
        transplant(T,z,z.right)
    ELSE
        IF z.right==nil THEN
            transplant(T,z,z.left)
        ELSE
            y=z.right;
            WHILE y.left != nil DO y=y.left;
            IF y.parent != z THEN
                transplant(T,y,y.right);
                y.right=z.right;
                y.right.parent=y;
            transplant(T,z,y);
            y.left=z.left;
            y.left.parent=y;
```

Laufzeit: $O(h)$, h ist die Höhe des Baumes

```
transplant(T,u,v)
    IF u.parent==nil THEN
        T.root=v
    ELSE
        IF u==u.parent.left THEN
            u.parent.left=v
        ELSE
            u.parent.right=v;
    IF v != nil THEN
        v.parent=u.parent;
```

Laufzeit: $\Theta(1)$
Zu löschender Knoten z hat maximal ein Kind:

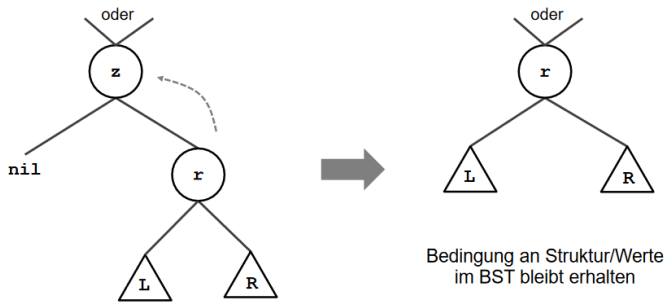


Figure 28: Visualisierung

Rechtes Kind von Knoten z hat kein linkes Kind:

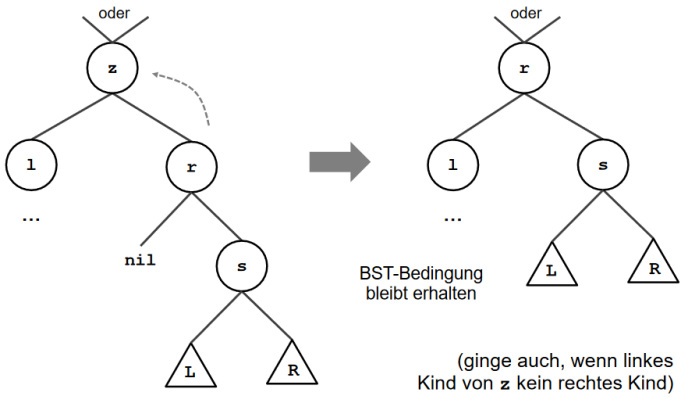


Figure 29: Visualisierung

'Kleinster' Nachfahre vom rechten Kind von z:

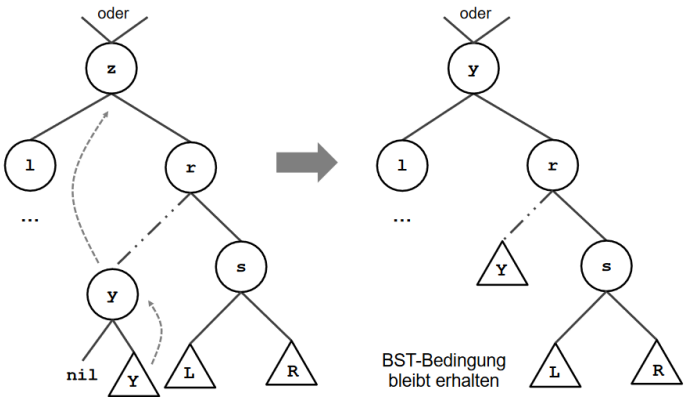


Figure 30: Visualisierung

Die Laufzeiten sind hierbei abhängig von der Höhe des BST.
Die Best- Case Laufzeit ist $O(log_2 n)$, wenn der BSt vollständig ist (alle Blätter haben gleiche Tiefe).
Die Worst-Case Laufzeit ist $\Omega(n)$, wenn der BST degeneriert ist (Baum besteht aus einer linearen Liste).

4 Advanced Data Structures

4.1 Rot- Schwarz- Bäume

Ein Rot-Schwarz-Baum ist ein binärer Suchbaum, so dass gilt:

- Jeder Knoten hat die Farbe rot oder schwarz
- Die Wurzel ist schwarz (sofern Baum nicht leer)
- Wenn ein Knoten rot ist, sind seine Kinder schwarz (Nicht- Rot- Rot-Regel)
- Für jeden Knoten hat jeder Pfad im Teilbaum zu einem Blatt oder Halbblatt die gleiche Anzahl von schwarzen Knoten (gleiche Anzahl schwarz)
- Hälbblätter sind schwarz

Dabei hat jeder Knoten noch einen besonderen Eintrag (x.color = red / black)
Beispiel:

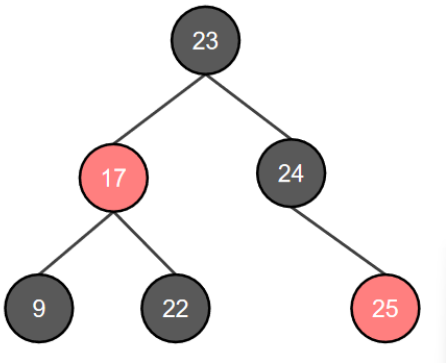


Figure 31: Ein Rot- Schwarz- Baum

Die Schwarzhöhe eines Knoten x ist die (eindeutige) Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt oder Halbblatt im Teilbaum des Knoten. Außerdem muss die Schwarzhöhe am Knoten für jeden Pfad gleich sein.

Einfügen:
Rotation:

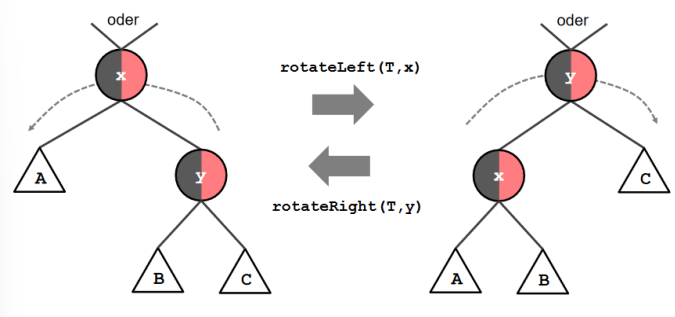


Figure 32: Rotation

```
rotateLeft(T,x) //x.right!=nil
    y=x.right;
    x.right=y.left;
    IF y.left != nil THEN
        y.left.parent=x;
    y.parent=x.parent;
    IF x.parent==T.sent THEN
        T.root=y
    ELSE
        IF x==x.parent.left THEN
            x.parent.left=y
        ELSE
            x.parent.right=y;
    y.left=x;
    x.parent=y;
```

Laufzeit: $\Theta(1)$.

```
insert(T,z) //z.left==z.right==nil;
    x=T.root; px=T.sent;
    WHILE x != nil DO
        px=x;
        IF x.key > z.key THEN
            x=x.left
        ELSE
            x=x.right;
    z.parent=px;
    IF px==T.sent THEN
        T.root=z
    ELSE
        IF px.key > z.key THEN
            px.left=z
        ELSE
            px.right=z;
    z.color=red;
```

```
18      fixColorsAfterInsertion(T,z);
```

```
1   fixColorsAfterInsertion(T,z)
2       WHILE z.parent.color==red DO
3           IF z.parent==z.parent.parent.left THEN
4               y=z.parent.parent.right;
5               IF y!=nil AND y.color==red THEN
6                   z.parent.color=black;
7                   y.color=black;
8                   z.parent.parent.color=red;
9                   z=z.parent.parent;
10              ELSE
11                  IF z==z.parent.right THEN
12                      z=z.parent;
13                      rotateLeft(T,z);
14                  z.parent.color=black;
15                  z.parent.parent.color=red;
16                  rotateRight(T,z.parent.parent);
17          ELSE
18              ... //exchange left and right
19      T.root.color=black;
```

Laufzeit: $O(n) = O(log_n)$
Schleifeninvariante:

- z.color==red
- Wenn z.parent Wurzel, dann z.parent.color==black
- Wenn der aktuelle Baum kein Rot-Schwarz-Baum ist, dann weil z als Wurzel die Farbe rot hat, oder weil 'Nicht-Rot-Rot-Regel' für z,z.parent verletzt ist. (anderen Regeln: Schwarzhöhe und jeder Knoten rot oder schwarz)

Löschen:
Analog zu binären Suchbäumen, aber der gelöschte Knoten übernimmt die Farbe des Nachrückers.
Wir gehen davon aus: z sei der gelöschte Knoten, y der Nachrücker. Falls y.color=black, müssen wir FixUp anwenden:
Fälle:

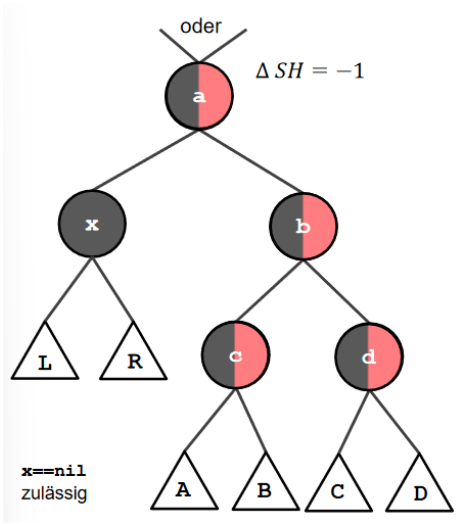


Figure 33: Zur Verdeutlichung

- Fall I: a schwarz, b rot (wird zu Fall IIa, III oder IV, nie zu IIb)
- Fall IIa: a rot, b schwarz, c,d nicht rot (Wenn a rot, dann auf schwarz setzen und damit ursprüngliche SH erreicht)
- Fall IIb: a schwarz, b schwarz, c,d nicht rot (Wenn a schwarz, dann Vaterknoten $\triangle SH = \pm 1$; verfahre rekursiv mit a als neuem x)
- Fall III: a beliebig, b schwarz, c rot, d nicht rot (wird zu Fall IV)
- Fall IV: a beliebig, b schwarz, c beliebig, d rot (ursprüngliche SH wie zuvor)

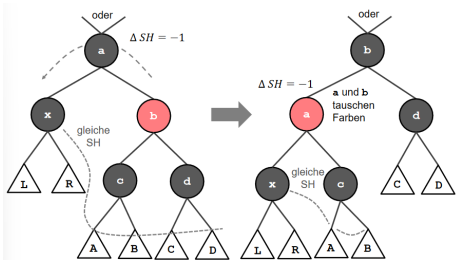


Figure 34: FixUp Fall I

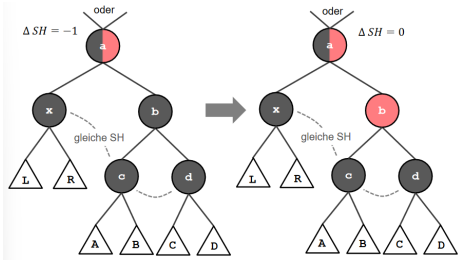


Figure 35: FixUp Fall IIa

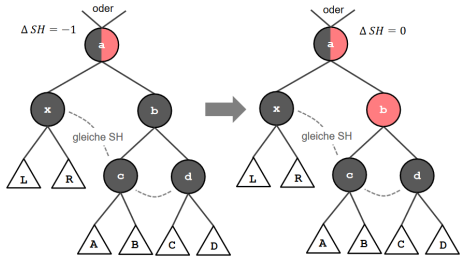


Figure 36: Zur VerdeFixUp Fall IIb

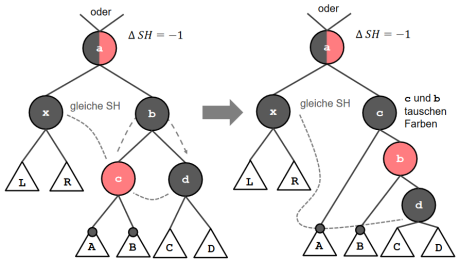


Figure 37: FixUp Fall III

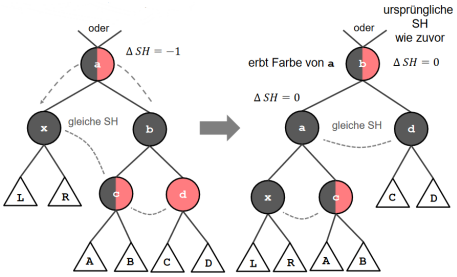


Figure 38: FixUp Fall IV

Laufzeit: $O(h) = O(log_n)$
Worst- Case- Laufzeiten:

- Einfügen: $\Theta(logn)$
- Löschen: $\Theta(logn)$
- Suchen: $\Theta(logn)$

4.2 AVL- Bäume

Benannt nach Georgi Maximowitsch Adelson-Velski und Jewgeni Michailowitsch Landis.
Ein AVL-Baum ist ein binärer Suchbaum, so dass für die Balance $B(x)$ in jede Knoten x gilt: $B(x) \in -1, 0, +1$.
Dabei ist $B(x) =$ Höhe$(rechterTeilbaum) -$ Höhe$(linkerTeilbaum)$.
Beispiel:

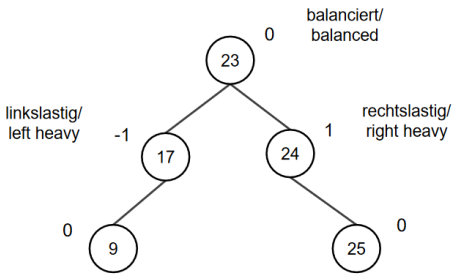


Figure 39: Beispiel

Höhe AVL- Baum: Sei n_h minimale Anzahl von Knoten in einem AVL-Baum der Höhe h. Allgemein gilt für die Berechnung $n_h = 1 + n_{h-1} + n_{h-2}$.

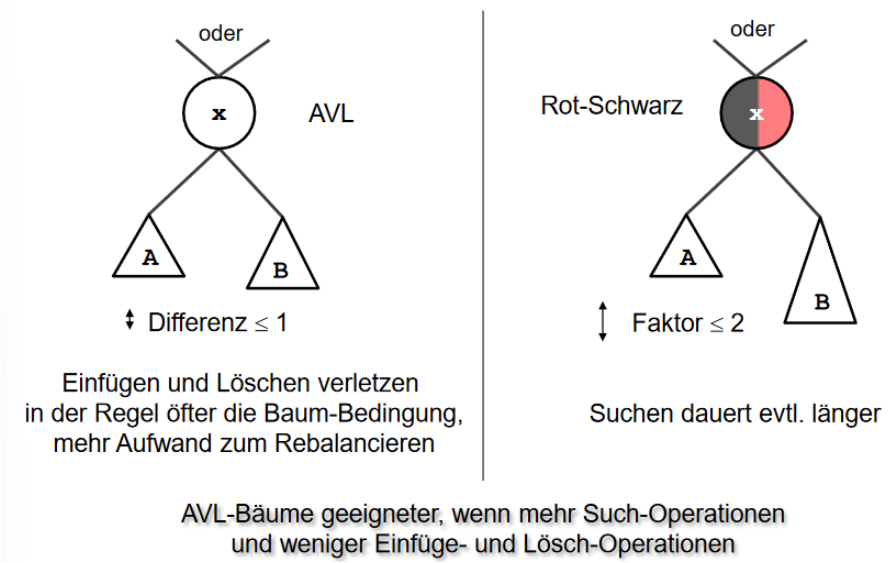


Figure 40: AVL vs. RS

Außerdem gilt: AVL $\subset$ Rot- Schwarz $\Rightarrow$ Jeder nicht-leere AVL-Baum der Höhe h lässt sich als Rot-Schwarz-Baum mit Schwarzhöhe $(h+1)/2$ darstellen.

```
1 insert(T,z) //z.left==z.right==nil;
2     x=T.root; px=T.sent;
3     WHILE x != nil DO
4         px=x;
5         IF x.key > z.key THEN
```

```
6               x=x.left
7           ELSE
8               x=x.right;
9       z.parent=px;
10      IF px==T.sent THEN
11          T.root=z
12      ELSE
13          IF px.key > z.key THEN
14              px.left=z
15          ELSE
16              px.right=z;
17      fixBalanceAfterInsertion(T,z);
```

Rebalancing:

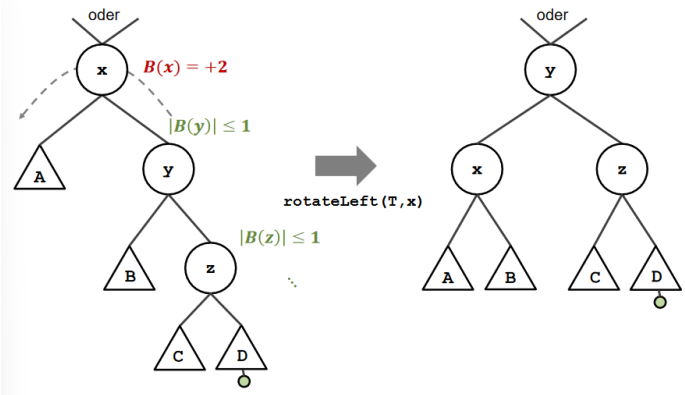


Figure 41: Fall 1

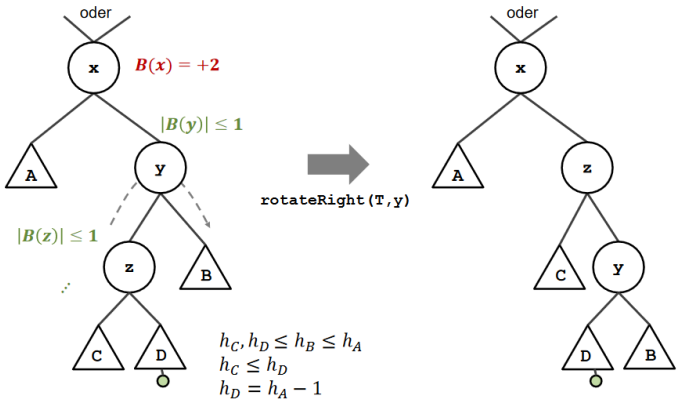


Figure 42: Fall 2, Teil 1

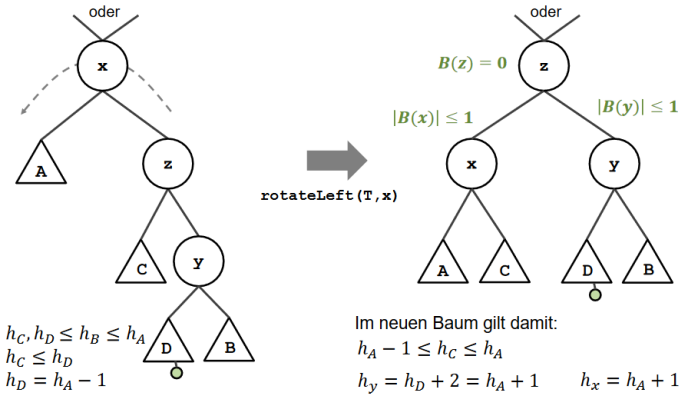


Figure 43: Fall 2, Teil 2

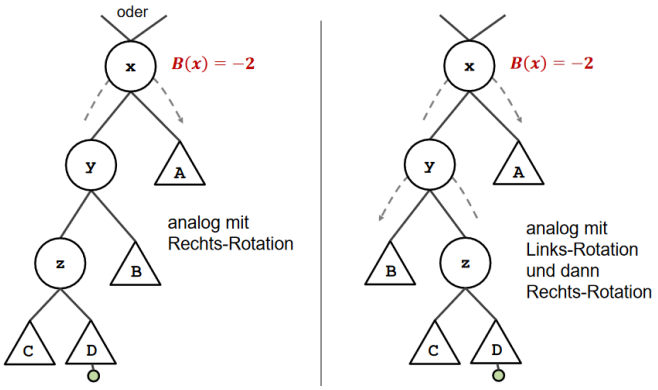


Figure 44: Fall 3 und 4

Löschen ist analog zu binären Suchbäumen, aber man muss das Rebalancing beachten.
Allgemein haben wir hier auch selbe Laufzeit wie bei Rot- Schwarz- Bäumen.

4.3 Splay- Bäume

Splay Bäume sind selbst- organisierte Datenstrukturen, die den gesuchten Knoten an die Position der Wurzel schieben.
Dabei gibt es die sogenannte Splay- Operation (**Laufzeit:** $O(h)$), die die Datenstruktur nach der oben genannten Regel verschiebt.

```
1 splay(T,z)
2     WHILE z != T.root DO
3         IF z.parent.parent==nil THEN
4             zig(T,z);
5         ELSE
```

```
6               IF z==z.parent.parent.left.left OR z==z.parent.
                    parent.right.right THEN
7                   zigZig(T,z);
8               ELSE
9                   zigZag(T,z);
```

Laufzeit: $O(h)$.
Operationen:

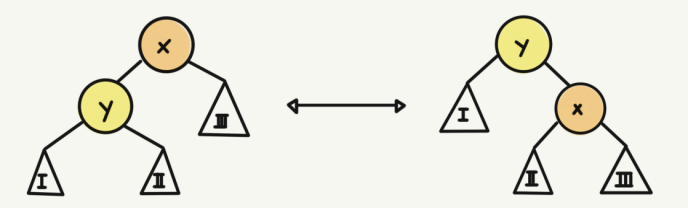


Figure 45: Zig- Operation

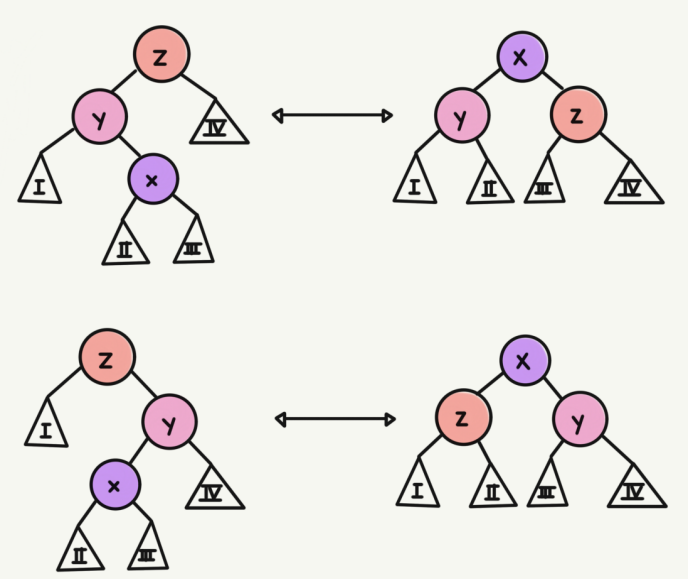


Figure 46: ZigZag- Operation

```
1   zigZig(T,z)
2       IF z==z.parent.left THEN
3           rotateRight(T,z.parent.parent);
4           rotateRight(T,z.parent);
5       ELSE
6           rotateLeft(T,z.parent.parent);
```

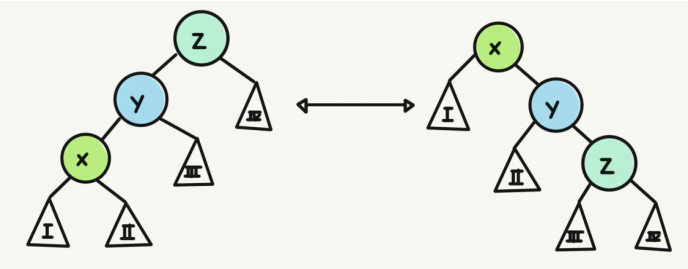


Figure 47: ZigZig- Operation

```
7           rotateLeft(T,z.parent);
```

Zig und ZigZag Operationen werden analog implementiert.
Suchen:

```
1  search(T,k)
2      x=T.root;
3      WHILE x != nil AND x.key != k DO
4          IF x.key < k THEN
5              x=x.right
6          ELSE
7              x=x.left;
8      IF x==nil THEN
9          RETURN nil
10     ELSE
11         splay(T,x);
12         RETURN T.root;
```

Laufzeit: $O(h)$.
Einfügen: Analog zum BST wird der Knoten eingefügt und danach mithilfe der Splay Operation nach oben gespült.
Laufzeit: $O(h)$.
Löschen: Per Splay- Operation nach oben spülen. Dann löschen (Falls einer der beiden Kinderteilbäume leer, ist man fertig, sonst:)

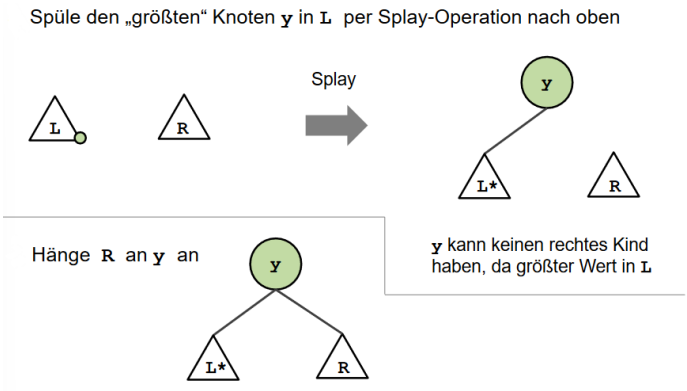


Figure 48: Verschieben nach Löschen

4.4 (Binäre Max-) Heaps

Ein binärer Max-Heap ist ein binärer Baum, der

- 'Bis auf das unterste Level vollständig und im untersten Level von links gefüllt ist'
- Für alle Knoten x $\neq$ T.root gilt: x.parent.key $\geq$ x.key

Achtung: Heaps sind keine BSTs, linke Kinder können größere Werte als rechte Kinder haben!

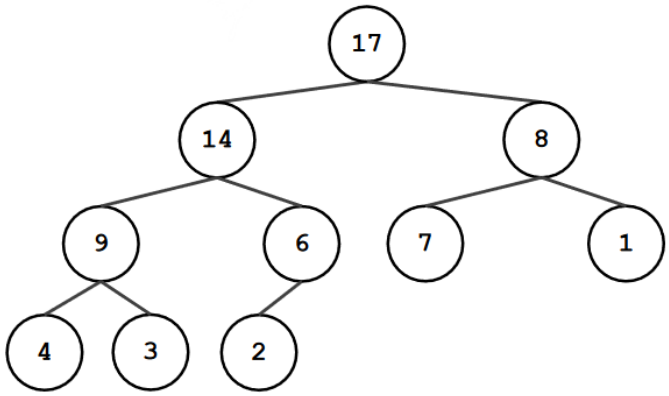


Figure 49: Beispiel für einen Heap

Ein Heap kann auch als Array dargestellt werden, in der von oben nach unten, von links nach rechhts in den Array eingefügt werden.

Einfügen:
Beim einfügen wird das Element an letzter Stelle eingfügt und so lange nach oben vertauscht, bis die Max- Eigenschaft wieder erfüllt ist.

```
insert(H,k) //als (unbeschraenktes) Array
    H.length=H.length+1;
    H.A[H.length-1]=k;
    i=H.length-1;
    WHILE i>0 AND H.A[i] > H.A[i.parent]
        SWAP(H.A,i,i.parent);
        i=i.parent;
```

Laufzeit: $O(h) = O(logn)$.
Löschen:
1. Erste Maximum durch 'letztes' Blatt
2.Stelle Max-Eigenschaften wieder her, indem Knoten nach unten gegen das Maximum der beiden Kinder getauscht wird (heapify)

```
extract-max(H) //als (unbeschraenktes) Array
    IF isEmpty(H) THEN
        RETURN error 'underflow';
    ELSE
        max=H.A[0];
        H.A[0]=H.A[H.length-1];
        H.length=H.length-1;
        heapify(H,0);
        RETURN max;
```

Laufzeit: $O(h) = O(logn)$.

```
heapify(H,i) //als (unbeschraenktes) Array
    maxind=i;
    IF i.left<H.length AND H.A[i]<H.A[i.left] THEN
        maxind=i.left;
    IF i.right<H.length AND H.A[maxind]<H.A[i.right] THEN
        maxind=i.right;
    IF maxind != i THEN
        SWAP(H.A,i,maxind);
        heapify(H,maxind);
```

Laufzeit: $O(h) = O(logn)$.
Heap- Konstruktion:

```
buildHeap(H) //Array A schon nach H.A kopiert
    H.length=A.length;
    FOR i = ceil((H.length-1)/2)-1 DOWNTO 0 DO
        heapify(H,i);
```

Laufzeit: $O(n \cdot h) = O(n \cdot logn)$.
Heap- Sort:

```
heapSort(H) //Array A schon nach H.A kopiert
    buildHeap(H);
    WHILE !isEmpty(H) DO PRINT extract-max(H) //Gibt
        Eintraege in Array A in absteigender Groesse aus
```

Laufzeit: $O(n \cdot h) = O(n \cdot logn)$.
In Java können diese mit Priority Queues dargestellt werden.

4.5 B- Bäume

Ein B-Baum (vom Grad t) ist ein Baum, bei dem

- Jeder Knoten (außer der Wurzel) zwischen t - 1 und 2t - 1 Werte key[0],key[1],... hat und die Wurzel zwischen 1 und 2t - 1
- Die Werte innerhalb eines Knoten aufsteigend geordnet sind
- Die Blätter alle die gleiche Höhe haben
- jeder innerer Knoten mit n Werten n + 1 Kinder hat, so dass für alle Werte k_j aus dem j-ten Kind gilt:
  $k_o \leq key[0] \leq k_1 \leq key[1] \leq ... \leq k_{n-1} \leq key[n-1] \leq k_n$.

Beispiel:

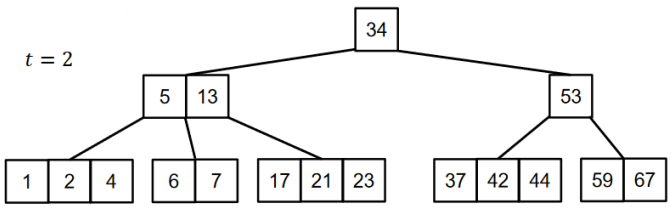


Figure 50: Beispiel B- Baum

Für die Höhe gilt:

- mindestens 1 Wert in Wurzel
- mindestens 2 Knoten in Tiefe 1 mit jeweils mindestens t Kindern
- mindestens 2t Knoten in nächster Tiefe mit jeweils mindestens t Kindern
- mindestens $2t^2$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern, usw.

Anzahl Werte n im B-Baum im Vergleich zu Höhe : $n \geq 1+(t-1)\cdot\sum_{i=1}^{h} 2t^{i-1} = 1+2(t-1)\cdot\frac{t^h-1}{t-1} = 2t^h-1 \Rightarrow log_t\frac{n+1}{2} \geq h$.
Ein B-Baum vom Grad t mit n Werten hat maximale Höhe $h \leq log_t\frac{n+1}{2}$.
Suchen:

```
search(x,k)
    WHILE x != nil DO
        i=0;
        WHILE i < x.n AND x.key[i] < k DO i=i+1;
        IF i < x.n AND x.key[i]==k THEN
            RETURN (x,i);
        ELSE
            x=x.child[i];
    RETURN nil;
```

Laufzeit: $O(t \cdot h) = O(log_t n)$.
Einfügen: Immer in einem Blatt. Wenn das Blatt dabei weniger als $2t-1$ Werte hat, sind wir fertig, sonst müssen wir splitten:
Sollte die Wurzel voll sein, wird diese gesplittet.

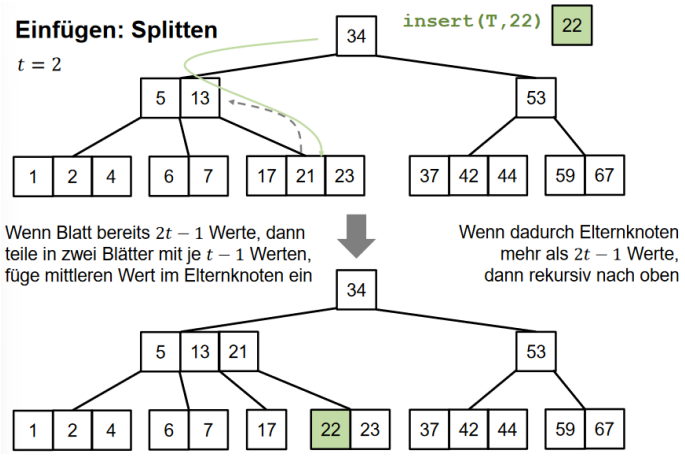


Figure 51: Splitten B- Baum

```
insert(T,z)
    Wenn Wurzel schon 2t-1 Werte, dann splitte Wurzel
    Suche rekursiv Einfuegeposition:
        Wenn zu besuchendes Kind 2t-1 Werte, splitte es erst
    Fuege z in Blatt ein
```

Löschen:
Löschen funktioniert intuitiv.

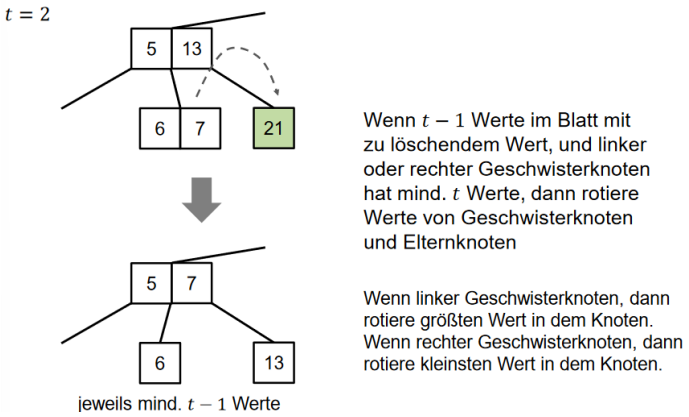


Figure 52: Löschen im 'zu leeren' Blatt: Rotieren

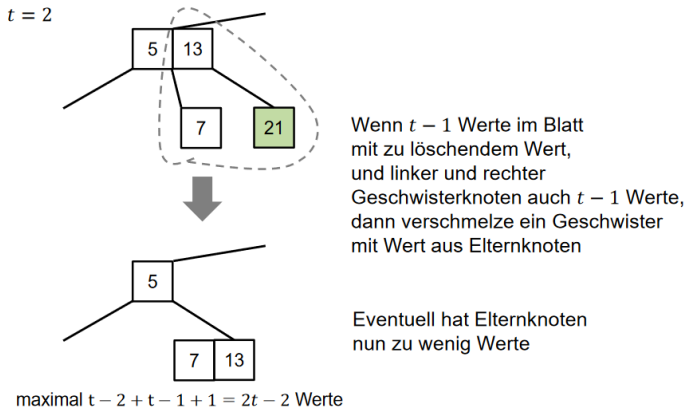


Figure 53: Löschen im 'zu leeren' Blatt: Verschmelzen

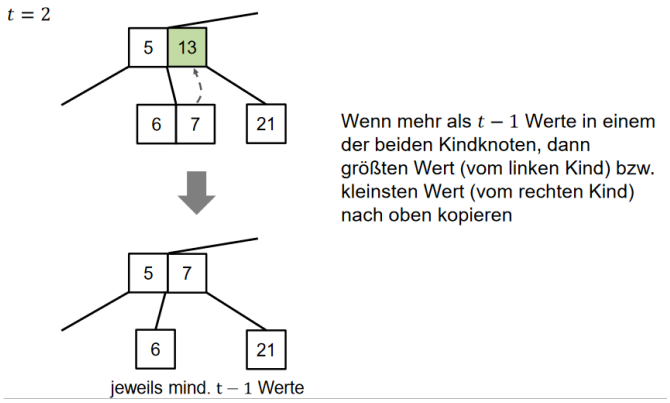


Figure 54: Löschen im inneren Knoten: Verschieben

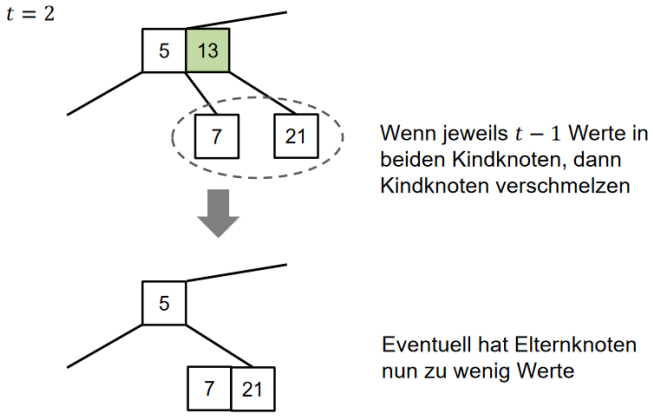


Figure 55: Löschen im inneren Knoten: Verschmelzen

```
1 delete(T,k)
2     Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,
          verschmelze Wurzel und Kinder (reduziert Hoehe um 1)
3     Suche rekursiv Loeschposition:
4         Wenn zu besuchendes Kind nur t-1 Werte, verschmelze
              es oder rotiere/verschiebe
5     Entferne Wert k in inneren Knoten/Blatt
```

Laufzeit: $O(t \cdot h) = O(log_t n)$.
Worst- Case- Laufzeiten:

- Einfügen/ Löschen/ Suchen: $\Theta(log_t n)$.

5 Randomized Data Structures

5.1 Skip- Lists

Eine Skip Liste ist eine Liste, in der eine 'Express- Liste' mit einigen Elementen ist.

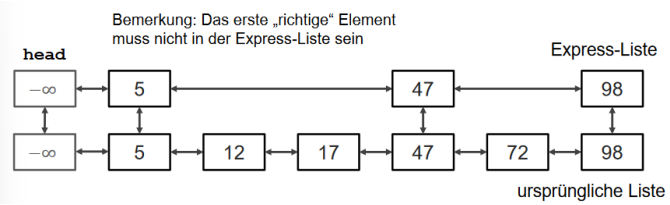


Figure 56: Beispiel einer Skip- Liste

Mann kann auch mehrere immer kleinere Express- Listen über der Express-Liste einbauen.
Suchen funktioniert hierbei nach dem folgenden Muster:

- Element gefunden? → Element ausgeben
- Nächstes Element in Express-Liste kleiner-gleich gesuchtes Element? → Weiter nach rechts
- Nächstes Element in Express-Liste größer als gesuchtes Element? → Nach unten in ursprüngliche Liste und dort weitersuchen

Dabei werden die Elemente auf jeder Express- Ebene zufällig mit einer Wahrscheinlichkeit von p gewählt. Implementierung:

- L.head - erstes/oberstes Element der Liste
- L.height - Höhe der Skiplist
- x.key - Wert
- x.next - Nachfolger
- x.prev - Vorgänger
- x.down - Nachfolger Liste unten
- x.up - Nachfolger Liste oben
- nil - kein Nachfolger / leeres Element

```
search(L,k)
    current=L.head;
    WHILE current != nil DO
        IF current.key == k THEN RETURN current;
        IF current.next != nil AND current.next.key <= k
        THEN current=current.next
        ELSE current=current.down;
    RETURN nil;
```

Durchschnittliche **Laufzeit:** $O(h)$, jedoch abhängig von der größe der Expresslisten.
Einfügen: In der untersten Ebene einfügen und dann per Wahrscheinlichkeit p in die darüber einfügen. (Durchschnittliche **Laufzeit:** $O(h)$)
Löschen: Entferne Vorkommen des Elements auf allen Ebenen (**Laufzeit:** $O(h)$)
Worst- Case- Laufzeiten (in Durchschnitt): $\Theta(log_{1/p}n)$.

5.2 Hash- Tables

Idee:

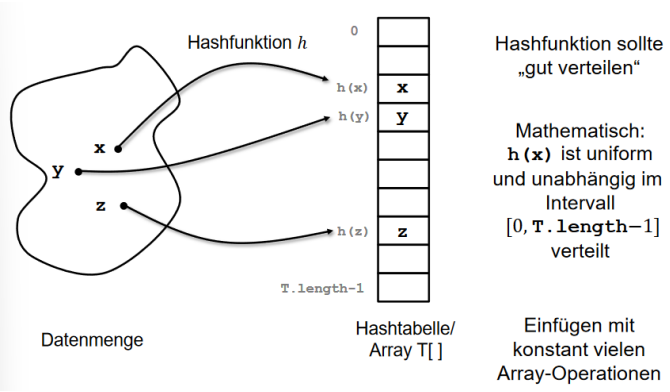


Figure 57: Grundkonzept eines Hash- Tables

Einfügen: Wir schauen, ob w in $T[h(w)]$ vorhanden ist (search(H,w))
Löschen: Intuitiv (delete(H,z))
Dabei erfolgt suchen/ löschen mit konstant vielen Array- Operation
Kolisionsauflösung: Falls ein Eintrag im Array schon belegt ist, bildet man eine verkettete Liste und fügt neues Element vorne ein.
Daraus folgt dann:

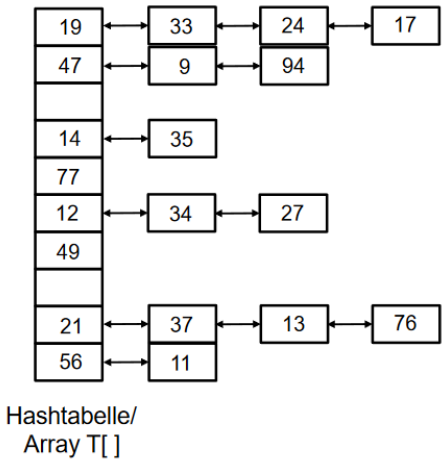


Figure 58: Hash- Tabelle als verkettete Liste

Einfügen immer noch konstante Anzahl Array-/Listen-Operationen.
Wenn Hashfunktion uniform verteilt, dann hat jede Liste im Erwartungswert $fracnT.length$ viele Einträge.
Worst- Case- **Laufzeiten:** (Im Durchschnitt)

- Einfügen: $\Theta(1)$
- Löschen: $\Theta(1)$
- Suchen: $\Theta(1)$

5.3 Bloom- Filter

Beispiel: Speichere ('offline') alle schlechten Passwörter im Bloom-Filter. Prüfe ('online'), ob eingegebenes Passwort im Bloom-Filter.
Anwendungsbeispiel: Bitcoin (Prüfen von Transaktionen); NoSQL- DB (Abfragen für nicht vorhandene ELemente verhindern).

Filter erstellen:
Gegeben:

- n Elemente $x_0 ..., x_{n-1}$beliebiger Komplexität
- m Bits Speicher, üblicherweise in einem Bit-Array
- k 'gute' Hash-Funktionen $H_0, ..., H_{k-1}$ mit Bildbereich $0, 1, ..., m-1$

Empfohlen wird jedoch: $k = \frac{m}{n} \cdot ln2$.

```
initBloom(X,BF,H) //H array of functions H[j]
FOR i=0 TO BF.length-1 DO BF[i]=0;
FOR i=0 TO X.length-1 DO
    FOR j=0 TO H.length-1 DO
        BF[H[j](X[i])]=1;
```

1. Inititalisierte Array mit 0 Einträgen.
2. Schreibe für jedes Element in jede Bit-Position $H_0(x_i), ..., H_{k-1}(x_i)$ eine 1.

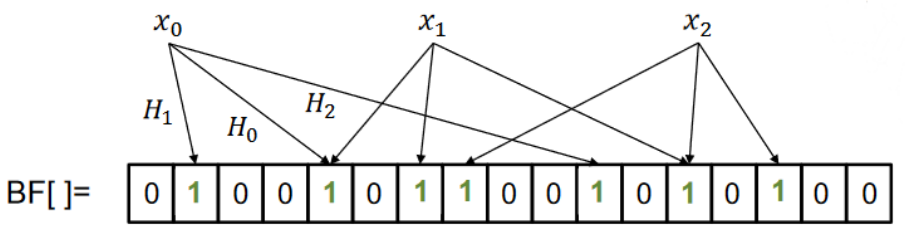


Figure 59: Visualisierung

Suchen:

```
searchBloom(BF,H,y) //H array of functions H[j]
    result=1;
    FOR j=0 TO H.length-1 DO
        result=result AND BF[H[j](y)];
    RETURN result;
```

Gibt an, dass y im Wörterbuch, wenn genau alle k Einträge für y in BF=1 sind.

6 Graph Algorithms

6.1 Graphen

(Endliche) Gerichtete Graphen:
Ein (endlicher) gerichteter Graph G = (V, E) besteht aus:

- einer (endlichen) Knotenmenge V ('vertices')
- einer (endlichen) Kantenmenge E $\subseteq$ V x V ('edges')

Dabei ist $(u, v) \in E$ Kante von Knoten u zu v.
Außerdem können keine Mehrfachkanten zwischen Knoten existieren.
Beispiel:

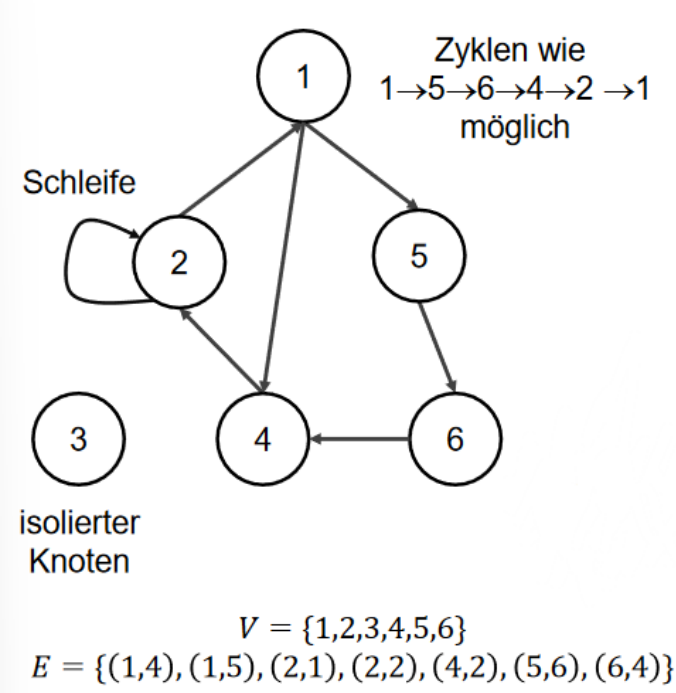


Figure 60: Beispiel

Ungerichtete Graphen:
Ein ungerichteter Graph G = (V, E) besteht aus:

- einer (endlichen) Knotenmenge V ('vertices')
- einer (endlichen) Kantenmenge E $\subseteq$ V x V ('edges'), so dass $(u, v) \in E \Leftrightarrow (v, u) \in E$

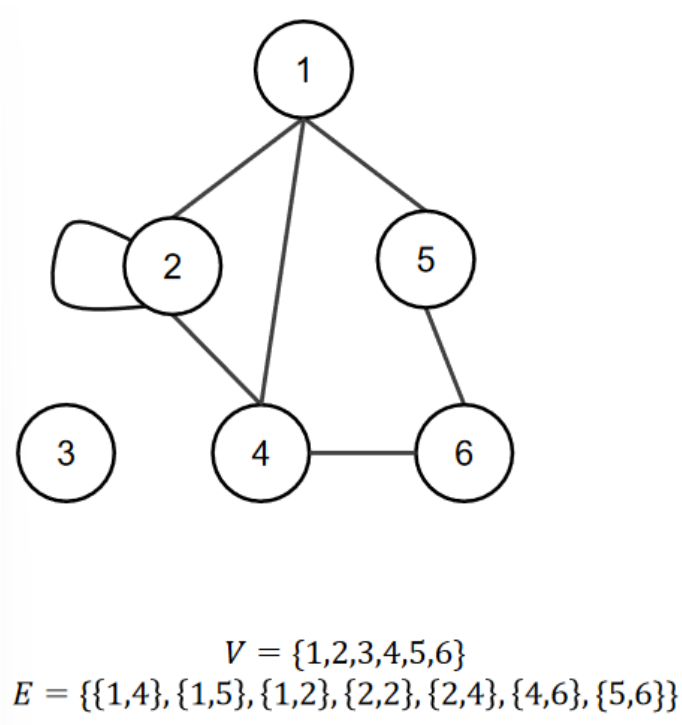


Figure 61: Beispiel

Man kann Graphen auch as $u, vstatt(u,v), (v,u)$ darstellen.

Dabei gilt: $shortest(u,v)$= Länge eines kürzestens Pfades von u nach v,
Zusammenhänge:

- Ungerichteter Graph ist **zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist
- Gerichteter Graph ist **stark zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus (gemäß Kantenrichtung) erreichbar ist

Graph G = (V, E) ist ein Baum, wenn V leer ist oder es einen Knoten $r \in V$ ('Wurzel') gibt, so dass jeder Knoten v von der Wurzel aus per eindeutigem Pfad erreichbar ist.
(Gerichteter oder ungerichteter) Graph G' = (V', E') ist Subgraph (oder Untergraph oder Teilgraph) des (gerichteten oder ungerichteten) Graphen G = (V, E), wenn V' $\subseteq$ V und E' $\subseteq$ E.
Man kann diese auch als Graphen darstelle (Speicherbedarf $\Theta(|V|^2)$).

Als Adjazenzmatrix:

$$A[i,j] = \begin{cases} 1 \text{ wenn Kante von i zu j} \\ 0 \text{ wenn keine Kante} \end{cases}$$

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

bei ungerichteten Graphen ist
Matrix (spiegel-)symmetrisch
zur Hauptdiagonalen

Figure 62: Matrix und Regeln

Matrixeigenschaften: Der Eintrag $a_{i,j}^{(m)}$ in der i-ten Zeile und j-ten Spalte der m-ten Potenz A^m der Adjazenzmatrix A eines Graphen gibt die Anzahl der Wege an, die von Knoten i zu Knoten j entlang von genau m Kanten führen (m $\geq$ 0).
Gewichteter Graph: Ein gewichteter gerichteter Graph G = (V, E) besitzt zusätzlich Funktion $w : E \to R$ Bei gewichteten ungerichteten Graphen gilt zusätzlich $w = ((u, v)) = w((v, u))$ für alle $(u, v) \in E$.

6.2 BFS (Breadth- First- Search)

Idee: Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn usw... Dabei bekommt jeder Knoten eine Farbe zugeordnet.

- WHITE = Knoten noch nicht besucht
- GRAY = in Queue für nächsten Schritt
- BLACK = fertig

```
1 BFS(G,s) //G=(V,E), s=source node in V
2     FOREACH u in V-{s} DO
3         u.color=WHITE; u.dist=+infinit; u.pred=NIL;
4     s.color=GRAY; s.dist=0; s.pred=NIL;
5     newQueue(Q);
6     enqueue(Q,s);
7     WHILE !isEmpty(Q) DO
```

```
        u=dequeue(Q);
        FOREACH v in adj(G,u) DO
            IF v.color==WHITE THEN
                v.color=GRAY; v.dist=u.dist+1; v.pred=u;
                enqueue(Q,v);
        u.color=BLACK;
```

Dabei ist *dist* = Distanz von s & *pred* = Vorgängerknoten & *adj(G,u)* = Liste aller Knoten $v \in V$ mit $(u, v) \in E$ (Rheinenfolge irrelevant).
Erklärung: Jeder Knoten hat zu Beginn keinen Vorgänger und Entfernung unendlich. Begonnen wird mit dem Starkknoten, der alle Knoten abläuft und sich selber als Vorgänger (mit der Entfernung) einträgt und diese zum Ablaufen einfärbt. Danach wird mit diesen Knoten weiter verfahren, bis die Queue Q leer ist.

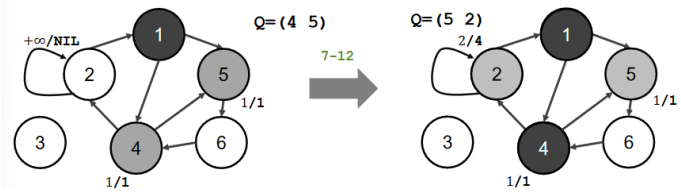


Figure 63: Besipiel für den Beginn des Algorithmus

Hierbei werden immer die abgearbeitet, die zuerst in der Queue waren. Bei Adding in der gleichen Iteration, wird zuerst das mit dem kleineren Index abgelaufen.
Laufzeit: $O(|V| + |E|)$, insgesamt werden maximal $\sum_{u \in V} |adj(G, u)| = O(|E|)$ Kanten betrachtet.
Algorithmus, um den kürzesten Pfad auszugeben:

```
PRINT-PATH(G,s,v) //assumes that BFS(G,s) has already been
    executed
    IF v==s THEN
        PRINT s;
    ELSE
        IF v.pred==NIL THEN
            PRINT 'no path from s -> v';
        ELSE
            PRINT-PATH(G,s,v.pred);
            PRINT v;
```

Laufzeit: $O(|V|)$ (ohne BFS).
Abgeleiteter BFS- Baum:
G^s_{pred} ist BFS- Baum zu G, d.h.enthält alle von s aus erreichbaren Knoten in

G und für jeden Knoten v$\in V^s_{pred}$ existiert genau ein Pfad von s in G^s_{pred}, der auch ein kürzester Pfad von s zu v in G ist. Definition:

- $G^s_{pred} = (V^s_{pred}, E^s_{pred})$
- $V^s_{pred} = \{v \in V | v.pred \neq NIL\} \cup s$
- $E^s_{pred} = \{(v.pred, v) | v \in V^s_{pred} - \{s\}$

(Für ungerichtete Graphen: $(v, v.pred)$)

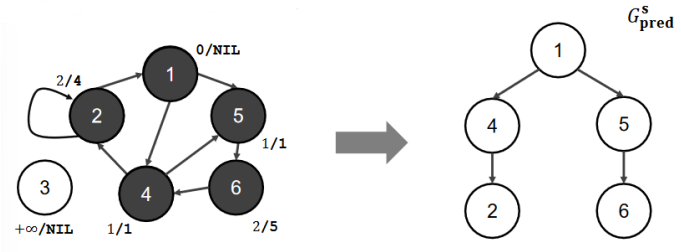


Figure 64: Beispiel für den Beginn des Algorithmus

6.3 DFS (Depth- First- Search)

Idee: Besuche zuerst alle noch nicht besuchten Nachfolgeknoten ('Laufe so weit wie möglich weg von aktuellem Knoten')

```
DFS(G) //G=(V,E)
    FOREACH u in V DO
        u.color=WHITE;
        u.pred=NIL;
    time=0;
    FOREACH u in V DO
        IF u.color==WHITE THEN
            DFS-VISIT(G,u);
```

Dabei ist *time* eine globale Variable.

```
DFS-VISIT(G,u)
    time=time+1;
    u.disc=time;
    u.color=GRAY;
    FOREACH v in adj(G,u) DO
        IF v.color==WHITE THEN
            v.pred=u;
            DFS-VISIT(G,v);
    u.color=BLACK;
    time=time+1;
    u.finish=time;
```

Dabei ist *disc* die discovery time und $finish$ die finish time.
Beispiel:

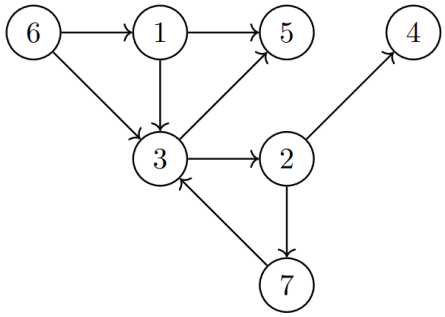


Figure 65: Der Baum für das Beispiel

Knoten	Entdeckungszeit	Abschlusszeit	Vorgängerknoten
1	1	12	*nil*
2	3	8	3
3	2	11	1
4	4	5	2
5	9	10	3
6	13	14	*nil*
7	6	7	2

Figure 66: Die daraus resultierende Tabelle

Abgeleiteter DFS- Baum:
Subgraph $G_{pred}(V, E_{pred})$ von G mit $E_{pred} = \{(v.pred, v) | v \in V, v.pred \neq NIL\}$ ((v,v.pred) für ungerichtete Graphen). (DFS Baum gibt dabei nicht unbegint den kürzesten Pfad wieder)
Kanten zeigen (Zeichne restlichen Kanten aus G auch in G_{pred} ein).

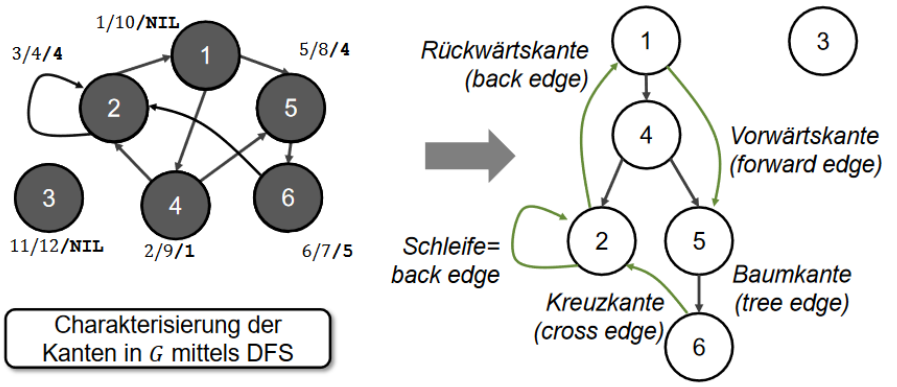


Figure 67: Kanten zeigen

Kantenart erkennen: Es gibt hierbei für gerichtete Graphen 4 verschiedene Fälle:
Sei (u, v) die gerade betrachtete Kante im DFS- Algorithmus. Dann ist (u, v)...

- ...**eine Baumkante, wenn v.color==WHITE** *(v noch nicht besucht, v.pred=u ⇒ (v.pred,v)=(u,v) als Kante in G_{pred})*
- ...**eine Rückwärtskante, wenn v.color==GRAY** *(die Kette von grauen Knoten, die auch im DFS-Baum eine Kette bilden)*
- ...**eine Vorwärtskante, wenn v.color==BLACK und u.disc<v.disc** *(da u.disc<v.disc wurde v schwarz, als u schon grau war; aber u ist noch nicht abgeschlossen, also wurde v von einem echten Nachkomme von u besucht, da u vorher zu einem weißen Knoten überging)*
- ...**eine Kreuzkante, wenn v.color==BLACK und v.disc<u.disc** *(alle anderen)*

6.3.1 Anwendung

BFS: Web Crawling, Kontakte in OSN, Broadcasting, Garbage Collection
DFS: Job Scheudling (In welcher Rheinenfolge sollen Jobs bearbeitet werden)
Topologische Sortierung eines dag $G = (V, E)$: Knoten in linearer Ordnung, so dass für alle Knoten $u, v \in V$ gilt, dass u vor v in der Ordnung kommt, wenn $(u, v) \in E$.

```
1 TOPOLOGICAL-SORT(G) // G=(V,E) dag
```

```
2     newLinkedList(L);
3     run DFS(G) but, each time a node is finished, insert in
          front of L;
4     RETURN L.head;
```

Laufzeit: $O(|V| + |E|)$.
Starke Zusammenhangskomponente: Eine starke Zusammenhangskomponente eines gerichteten Graphen $G = (V, E)$ ist eine Knotenmenge $C \subseteq V$, so dass
(a) es zwischen je zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt, und
(b) es keine Menge $D \subseteq V$ mit $C! \subseteq D$ gibt, für die (a) auch gilt
(C ist maximal).

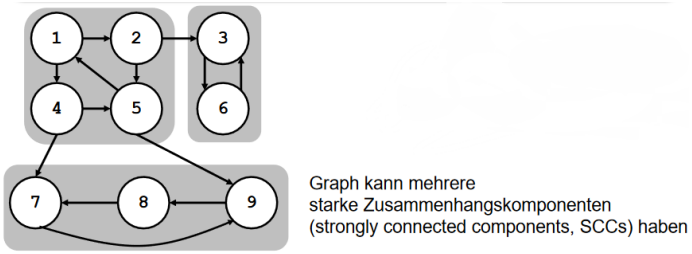


Figure 68: Starke Zusammenhangskomponente Visualisierung

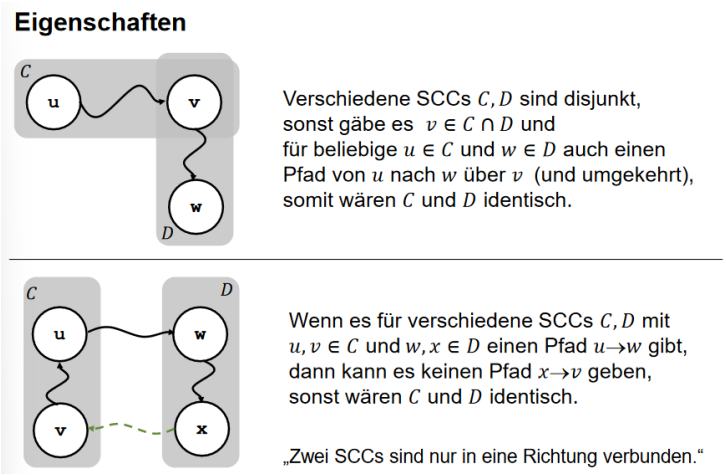


Figure 69: Eigenschaften

Lasse zweimal DFS laufen:

- einmal auf Graph G

- einmal auf transponiertem Graphen $G^T = (V, E^T)$ ($E^T = \{(v,u)|(u,v) \in E\}$)

SCCs in G und G^T bleiben identisch (in beiden Fällen gibt es in jeder SCC einen Weg von jedem Knoten zum anderen Knoten), nur Übergänge zwischen SCCs drehen sich um.

```
1 SCC(G) // G=(V,E) directed graph
2     run DFS(G)
3     compute G^T
4     run DFS(G^T) but visit vertices in main loop in
          descending finish time from 1
5     output each DFS tree in 3 as one SCC
```

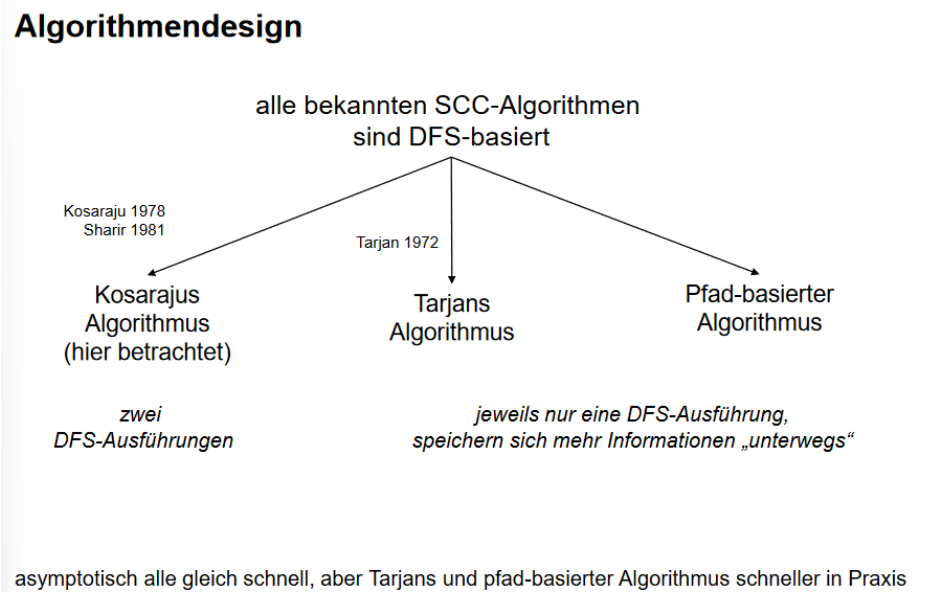


Figure 70: Eigenschaften

6.4 Minimale Spannbäume (MST)

Für einen zusammenhängenden, ungerichteten, gewichteten Graphen $G = (V, E)$ mit Gewichten w ist der Subgraph $T = (V, E_T)$ von G ein Spannbaum ('spanning tree'), wenn T azyklisch ist und alle Knoten verbindet.
Der Spannbaum ist minimal, wenn $w(T) = \sum_{\{u,v\} \in E_T} w(\{u,v\})$ minimal für alle Spannbäume von G ist.
Idee:

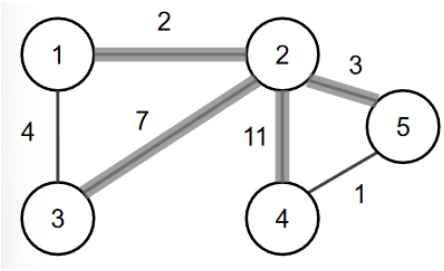


Figure 71: Beispiel

```
1 genericMST(G,w) // G=(V,E) undirected, connected graph w
      weight function
2     A={};
3     WHILE A does not form a spanning tree for G DO
4         find safe edge {u,v} for A;
5         A = A vereinigt {{u,v}};
6     RETURN A;
```

A: Teilmenge der Kanten eines MST; Kante $\{u, v\}$ ist sicher ('safe') für A, wenn $A \cup \{\{u, v\}\}$ noch eine Teilmenge eines MST ist.
Terminierung: Da wir zeigen werden, dass es in jeder Iteration eine sichere Kante für A gibt (sofern A noch kein Spannbaum), terminiert die Schleife nach maximal $|E|$ Iterationen.
Korrektheit: Da in jeder Iteration nur sichere Kanten hinzugefügt werden (für die $A \cup \{\{u, v\}\}$ noch Teilmenge eines MST ist), ist am Ende der WHILE-Schleife A ein MST.
'Leicht=sicher' legt Greedy- Strategie für konkrete Implementierung nahe. Da-

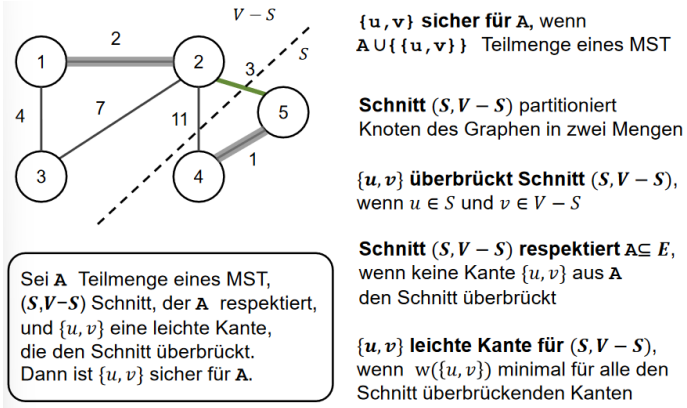


Figure 72: Terminologie udn Leicht=Sicher?

raus folgen zwei Ansätze:

- Algorithmus von Kruskal (lässt parallel mehrere Unterbäume eines MST wachsen)
- Algorithmus von Prim (konstruiert einen MST Knoten für Knoten)

6.4.1 Kruskal

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph w
    weight function
    A{};
    FOREACH v in V DO set(v)={v};
    Sort edges according to weight in nondecreasing order;
    FOREACH {u,v} in E according to order DO
        IF set(u)!=set(v) THEN
            A = A vereinigt {{u,v}}
            UNION(G,u,v);
    RETURN A;
```

Dabei setzt $Union(G, u, v)$ für alle Knoten $w \in set(u) \cup set(v)$: $set(w) = set(u) \cup set(v)$.
Zu jedem Knoten v sei $set(v)$ Menge von mit v durch A verbundenen Knoten.
Zu Beginn ist $set(v) = \{v\}$. $(set(u), set(v)$ sind disjunkt oder identisch).
Beispiel:

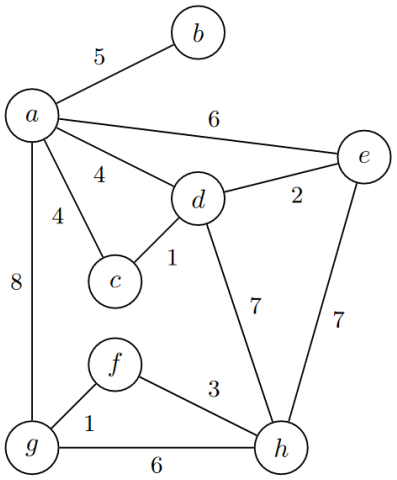


Figure 73: Beispiel

$\{u,v\}$	$w(\{u,v\})$	Dazu?	set(a)	set(b)	set(c)	set(d)	set(e)	set(f)	set(g)	set(h)
□	□	□	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
$\{c,d\}$	1	j	=	=	$\{c,d\}$	$\{c,d\}$	=	=	=	=
$\{f,g\}$	1	j	=	=	=	=	=	$\{f,g\}$	$\{f,g\}$	=
$\{d,e\}$	2	j	=	=	$\{c,d,e\}$	$\{c,d,e\}$	$\{c,d,e\}$	=	=	=
$\{f,h\}$	3	j	=	=	=	=	=	$\{f,g,h\}$	$\{f,g,h\}$	$\{f,g,h\}$
$\{a,c\}$	4	j	$\{a,c,d,e\}$	=	$\{a,c,d,e\}$	$\{a,c,d,e\}$	$\{a,c,d,e\}$	=	=	=
$\{a,d\}$	4	n	=	=	=	=	=	=	=	=
$\{a,b\}$	5	j	$\{a,b,c,d,e\}$	$\{a,b,c,d,e\}$	$\{a,b,c,d,e\}$	$\{a,b,c,d,e\}$	$\{a,b,c,d,e\}$	=	=	=
$\{a,e\}$	6	n	=	=	=	=	=	=	=	=
$\{g,h\}$	6	n	=	=	=	=	=	=	=	=
$\{d,h\}$	7	j	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$	$\{a,b,c,d,e,f,g,h\}$
$\{e,h\}$	7	n	=	=	=	=	=	=	=	=
$\{a,g\}$	8	n	=	=	=	=	=	=	=	=

Figure 74: Tabelle dazu

Laufzeit: $O(|E| \cdot log(|E|))$ (mit vielen Optimierungen)

6.4.2 Prim

```
1  MST-Prim(G,w,r) // r root in V, MST given through v.pred
       values
2      FOREACH v in V DO {v.key=infinit; v.pred=NIL;}
3      r.key=-infinite; Q=V;
4      WHILE !isEmpty(Q) DO
5          u=EXTRACT-MIN(Q); //smallest key value
6          FOREACH v in adj(u) DO
7              IF v in Q and w({u,v})<v.key THEN
8                  v.key=w({u,v});
9                  v.pred=u;
```

Idee: Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zu zusammenhängender Menge hinzu. Auswahl der nächsten Kante gemäß key-Wert, der stets aktualisiert wird.
A implizit definiert durch $A = \{\{v, v.pred\} | v \in V - (\{r\} \cup Q)\}$.
Beispiel:

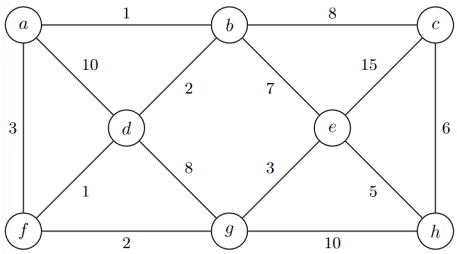


Figure 75: Beispiel

$a.k$	$b.k$	$c.k$	$d.k$	$e.k$	$f.k$	$g.k$	$h.k$	u
$-\infty$	∞	∞	∞	∞	∞	∞	∞	–
=	1	=	10	=	3	=	=	a
=	=	8	2	7	=	=	=	b
=	=	=	=	=	1	8	=	d
=	=	=	=	=	=	2	=	f
=	=	=	=	3	=	=	10	g
=	=	=	=	=	=	=	5	e
=	=	6	=	=	=	=	=	h
=	=	=	=	=	=	=	=	c

$a.p$	$b.p$	$c.p$	$d.p$	$e.p$	$f.p$	$g.p$	$h.p$	Q
nil	nil	nil	nil	nil	nil	nil	nil	$\{a,b,c,d,e,f,g,h\}$
=	a	=	a	=	a	=	=	$\{b,c,d,e,f,g,h\}$
=	=	b	b	b	=	=	=	$\{c,d,e,f,g,h\}$
=	=	=	=	=	d	d	=	$\{c,e,f,g,h\}$
=	=	=	=	=	=	f	=	$\{c,e,g,h\}$
=	=	=	=	g	=	=	g	$\{c,e,h\}$
=	=	=	=	=	=	=	e	$\{c,h\}$
=	=	h	=	=	=	=	=	$\{c\}$
=	=	=	=	=	=	=	=	$\{\}$

Figure 76: Tabelle dazu

Laufzeit: $O(|E|+|V|\cdot log(|V|))$ (mit vielen Optimierungen, speziell Fibonacci-Heaps)

6.5 SSSP (Kürzester Weg)

Finde von Quelle s aus jeweils den (gemäß Kantengewichten) kürzesten Pfad zu allen anderen Knoten.

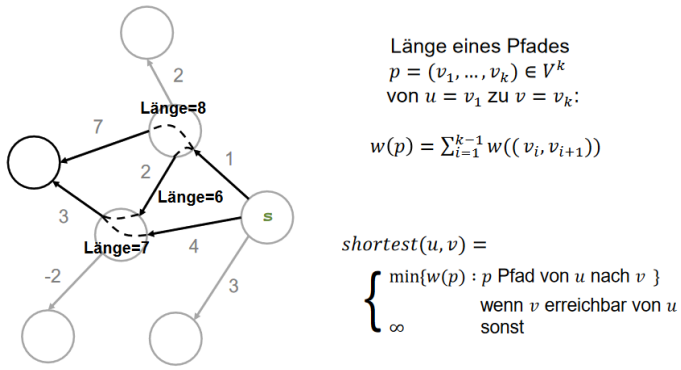


Figure 77: Erklärung

Der Unterschied zu den anderen Graphalgorithmen liegt darin, dass BFS und DFS keine Kantengewichte haben, also suchen sie nach dem min. Kantenweg und nicht dem min. Gewichtsweg. Ein MST (für ungerichteten Graphen) minimiert Gesamtgewicht $w(T) = \sum w(\{u, v\})$ des Baumes.
Dabei sind Zyklen erlaubt, aber keine (erreichbaren) Zyklen mit negativem Gesamtgewicht, da in diesem Fall der Durchlauf eine beliebig kleine Gesamtlänge ergeben würde.
Kürzeste Teilpfade: Teilpfad $s \to x$ eines kürzesten Pfades $s \to x \to z$ ist auch stets kürzester Pfad von s nach x (sonst gäbe es kürzeren Pfad von s nach z)

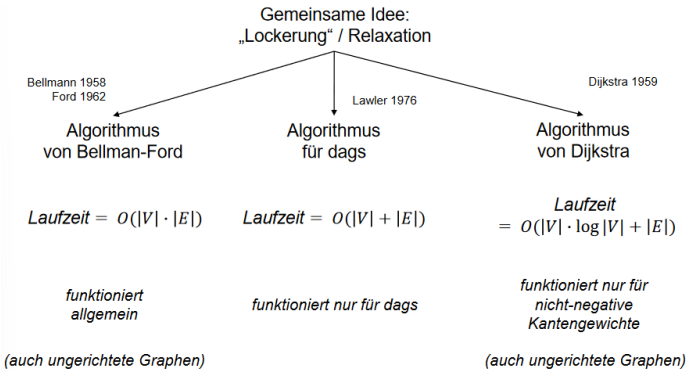


Figure 78: Übersicht

Relax: Idee: verringere aktuelle Distanz von Knoten v, wenn durch Kante (u, v) kürzere Distanz erreichbar (Zu Beginn: $\forall Knoten \setminus s : Distanz = \infty$):

```
1 relax(G,u,v,w)
2     IF v.dist > u.dist + w((u,v)) THEN
```

```
3           v.dist=u.dist + w((u,v));
4           v.pred=u;
```

6.5.1 Bellmann- Ford

```
1  Bellman-Ford-SSSP(G,s,w)
2      initSSSP(G,s,w);
3      FOR i=1 TO |V|-1 DO
4          FOREACH (u,v) in E DO
5              relax(G,u,v,w);
6      FOREACH (u,v) in E DO
7          IF v.dist > u.dist+w((u,v)) THEN
8              RETURN false;
9      RETURN true;
```

Dieser Algorithmus prüft zusätzlich, ob 'negativer Zyklus' erreichbar (=false).
Laufzeit: $\Theta(|E| \cdot |V|)$ (wegen geschachtelter FOR-Schleifen in 3 und 4)

```
1  initSSSP(G,s,w)
2      FOREACH v in V DO
3          v.dist=infinit;
4          v.pred=NIL;
5      s.dist=0;
```

Der Algorithmus läuft Intuitiv, am Ende würde der Graph so aussehen:

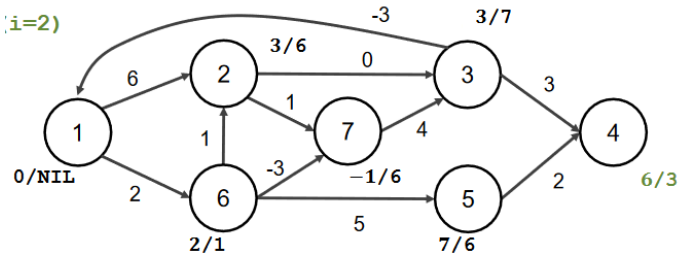


Figure 79: Ende eines Beispiels

6.5.2 TopoSort

```
1  TopoSort-SSSP(G,s,w) // G dag
2      initSSSP(G,s,w);
3      execute topological sorting
4      FOREACH u in V in topological order DO
5          FOREACH v in adj(u) DO
6              relax(G,u,v,w);
```

Dieser Algorithmus geht jeden Knoten durch und passt die Werte an.
Beispiel für einen TopoSort- Graphen (vor der Sortierung):

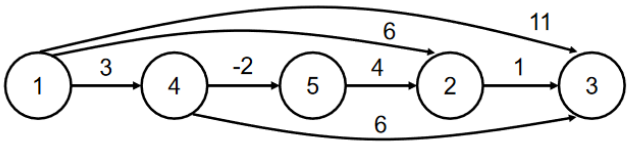


Figure 80: Beispielgraph

Laufzeit: $\Theta(|E| + |V|)$

6.5.3 Dijkstra

Vorraussetzung für diesen Algorithmus: $\forall Kanten : w((u, v)) \geq 0$

```
Dijkstra-SSSP(G,s,w)
    initSSSP(G,s,w);
    Q=V; //let S=V-Q
    WHILE !isEmpty(Q) DO
        u=EXTRACT-MIN(Q); //wrt. dist
        FOREACH v in adj(u) DO
            relax(G,u,v,w);
```

Laufzeit: $\Theta(|V| \cdot log|V| + |E|)$

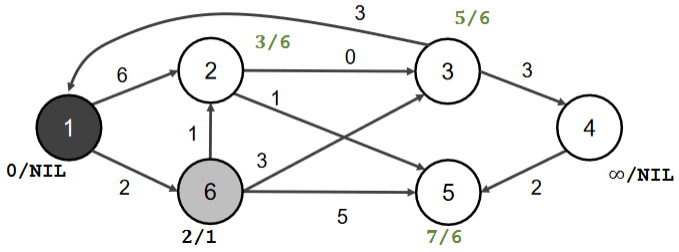


Figure 81: Beispiel während des Algorithmus eines Beispielgraphen

Jedoch funktioniert dieser Algorithmus mit negativen Kantengewichten nur sehr schwierig bis gar nicht.

6.5.4 A*- Algorithmus

Spezialfall: suche kürzesten Weg von s zu einem Ziel t.
Idee: Füge Heuristik hinzu, die 'vom Ziel her denkt'.

```
A*(G,s,t,w)
    init(G,s,t,w);
    Q=V; //let S=V-Q
    WHILE !isEmpty(Q) DO
        u=EXTRACT-MIN(Q);
        IF u==t THEN break;
        FOREACH v in adj(u) DO
            relax(G,u,v,w);
```

Jeder Knoten u bekommt zusätzlich Wert u.heur zugewiesen (Beispiel: Abstand Luftlinie vom Ziel)

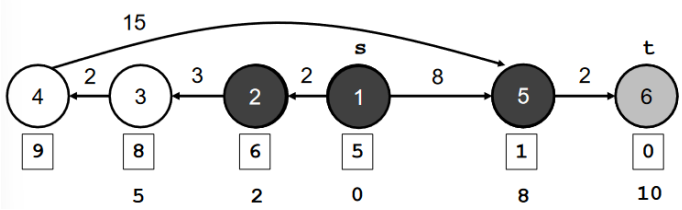


Figure 82: Visualisierung

Dabei sucht der Algorithmus erst in falscher Richtung.
Dies funktioniert aber nur mit nicht-negativen Kantengewichten:
A* findet optimale Lösung, wenn gilt:

- Heuristik überschätzt nie tatsächliche Kosten: $u.heur \leq shortest(u,t)$
- Heuristik ist monoton, d.h. für alle $(u,v) \in E$ gilt: $u.heur \leq w(u,v) + v.heur$

'Dijkstra ist A* mit Heuristik 0'. 'A* mit monotoner Heuristik ist Dijkstra mit Kantengewichten w(u,v)+v.heur-u.heur und s.dist=s.heur'

6.6 Netzwekflüsse (Max. Fluss)

Idee:

- Kanten haben (aktuellen) Flusswert und (maximale) Kapazität
- Jeder Knoten außer s und t hat gleichen eingehenden und ausgehenden Fluss
- Ziel: Finde maximalen Fluss von s nach t

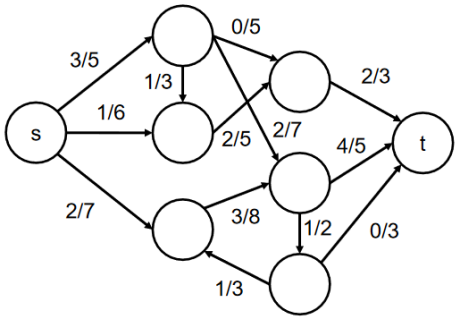


Figure 83: Beispiel eines Graphen mit Netzwerkflüssen

Netzwerkflüsse:
Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität(-sgewicht) c, so dass $c(u, v) \geq 0$ für $(u, v) \in E$ und $c(u, v) = 0$ für $(u, v) \notin E$, mit zwei Knoten $s, t \in V$ (Quelle und Senke), so dass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist.
Ein Fluss $f : V \times V \to R$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$, sowie für alle $u \in V - \{s, t\}$: $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

Maximale Flüsse:
Der Wert $|f|$ eines Flusses $f : V \times V \to R$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t ist $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$.

Transformation:

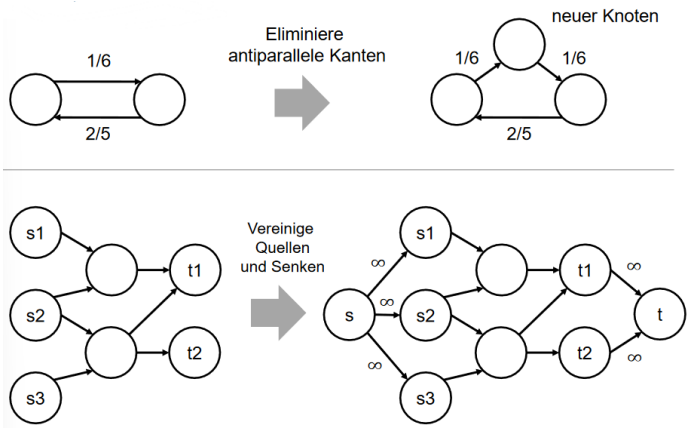


Figure 84: Transformation

Idee Ford- Fulkerson- Methode Idee: Suche Pfad von s nach t, der noch erweiterbar (bzgl. des Flusses) ist.
Aber: Pfad suchen wir im 'Restkapazitäts'-Graph G_f, der die möglichen Zu- und Abflüsse beschreibt.
Restkapazitäten:
Hierbei gilt: $c_f(u,v) =$

- $c(u,v) - f(u,v)$ falls $(u,v) \in E$ (Wieviel eingehenden Fluss über (u,v) könnte man noch zu v hinzufügen)
- $f(v,u)$ falls $(v,u) \in E$ (Wieviel abgehenden Fluss über (v,u) könnte man wegnehmen und damit quasi zu v hinzufügen?)
- 0 sonst

Für den Restkapazitätengraph gilt: $G_f = (V, E_f)$ mit $E_f = \{(u,v) \in V \times V | c_f(u,v) > 0\}$.
Finde Pfad von s zu t in G_f und erhöhe (für Kanten in G) bzw. erniedrige (für Nicht-Kanten) um Minimum $c_f(u,v)$ aller Werte auf dem Pfad in G

6.6.1 Ford- Falkeson- Algorithmus

```
1 Ford-Fulkerson(G,s,t,c)
2     FOREACH e in E DO e.flow=0;
3     WHILE there is path p from s to t in G_flow DO
4         c_flow(p) = min{c_flow(u,v)|(u,v) in p}
5         FOREACH e in p DO
6             IF e in E THEN
7                 e.flow=e.flow + c_flow(p)
8             ELSE
9                 e.flow=e.flow - c_flow(p)
```

Beispiel:

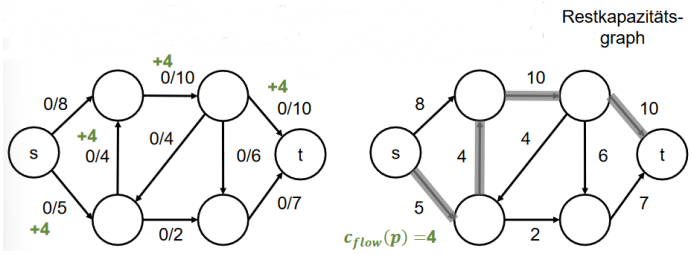


Figure 85: Start eines Beispiels

Max-Flow Min-Cut Theorem: Sei $f : V \times V \to R$ Fluss über ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t. Dann sind äquivalent:

- f ist ein maximaler Fluss für G
- Der Restkapazitätengraph G_f enthält keinen erweiterbaren Pfad
- $|f| = min_s c(S, V - S)$ mit $s \in S$ und $t \in V - S$

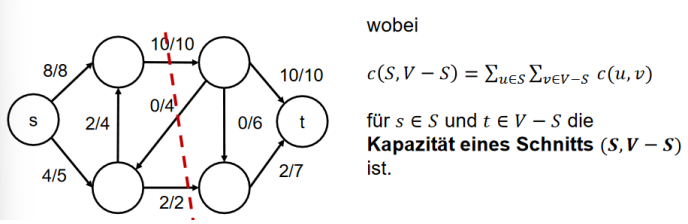


Figure 86: Cut- Beispiel

Beispielanwendung:

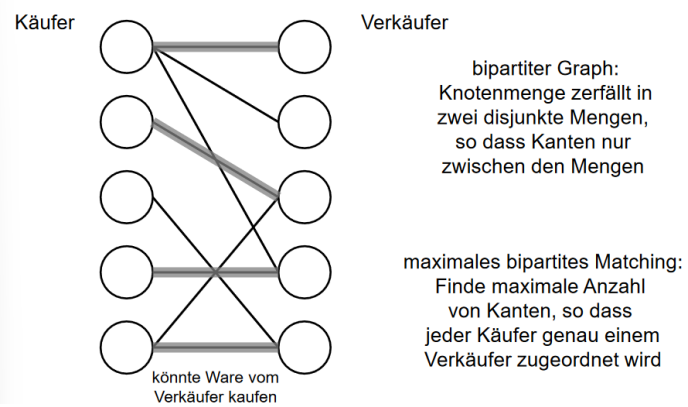


Figure 87: Beispiel

7 Advanced Designs

Es gibt verschiedene Entwurfsmethoden:

- **Divide** & **Conquer**: Löse reskursiv (disjunkte) Teilbäume, z.B. Quicksort und Mergesort
- **Backtracking**: Durchsuche iterativ Lösungsraum
- **Dynamische Programmierung**: Löse reskursiv (überlappende) Teilprobleme durch Wiederverwenden
- **Greedy**: Baue Lösung aus Folge lokal bester Auswahlen zusammen, z.B. Kruskal, Prim und Dijkstra
- (**Metaheuristiken**: übergeordnete Methoden für Optimierungsprobleme)

7.1 Backtracking

Prinzip: Finde Lösungen $x = (x_1, x_2, \ldots, x_n)$ per 'Trial-and-Error', indem Teillösung $(x_1, x_2, \ldots, x_{i-1})$ durch Kandidaten x_i ergänzt wird, bis Gesamtlösung erhalten, oder bis festgestellt, dass keine Gesamtlösung erreichbar, und Kandidat x_{i-1} revidiert wird.
Beispiel: Sudoku

```
1  SUDOKU-BACKTRACKING(B) // B[0...3][0...3] board
2      IF isFull(B) THEN
3          print 'solution: '+B;
4      ELSE
5          (i,j)=nextFreePos(B);
6          FOR v=1 TO 4 DO
7              IF isAdmissible(B,i,j,v) THEN
8                  B[i,j]=v;
9                  SUDOKU-BACKTRACKING(B);
10         B[i,j]=empty;
```

- *Zeile 2: prüft, ob Board komplett ausgefüllt*
- *Zeile 5: nächste freie Position (zeilenweise)*
- *Zeile 7: prüft Kriterien oben*
- *Zeile 10: löscht feld vor Rückkehr*

Backtracking funktioniert in diesem Fall, indem es von allen möglichen Zahlen eine auswählt, diese einträgt und weitermacht und falls es nicht mehr funktioniert, eine zurückgeht und eine andere Zahl einsetzt.
BT vs. DFS: Backtracking kann man als Tiefensuche auf Rekursionsbaum betrachten, wobei aussichtslose Lösungen evtl. frühzeitig abgeschnitten werden.

BT vs. Brute- Force: Backtracking kann man ist 'intelligentere' erschöpfende Suche ansehen, die aussichtslose Lösungen vorher aussortiert.
Lösungssuche: Man kann entweder eine Lösung, alle Lösungen oder die beste Lösung finden.
Beispiel: Regulärer Ausdruck: $a(b|c)+d \rightarrow$ ...mit a beginnt ...dann ein oder mehrere Zeichen aus der Menge $\{b, c\}$ enthält ...und noch auf d endet $\Rightarrow$ durchsuche String abacd per Backtracking 'gegen' regulären Ausdruck

7.2 Dynamische Programmierung

Prinzip: Teile Problem in (überlappende) Teilprobleme. Löse rekursiv Teilprobleme, verwende dabei Zwischenergebnisse wieder ('Memoization'). Rekonstruiere Gesamtlösung.
Beispiel: Fibonacci-Zahlen

```
1 Fib-Rek(n) // n>=1
2     IF n=<2 THEN
3         RETURN 1;
4     ELSE
5         RETURN Fib-Rek(n-1)+Fib-Rek(n-2);
```

(Vereinfachte) **Laufzeitabschätzung:** $\Theta(2^n)$.
Fibonacci- Berechnungsbaum:

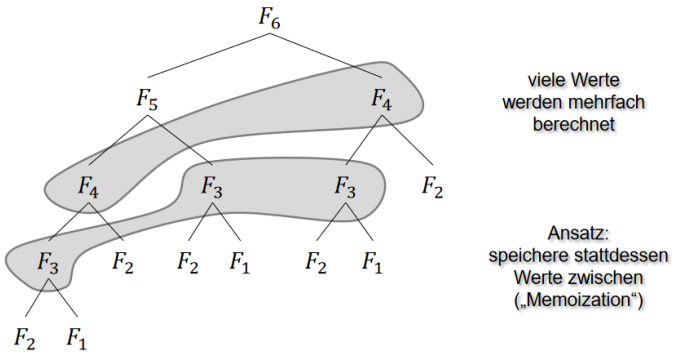


Figure 88: Berechnungsbaum

Fibonacci mit Memoization:

```
1 FibDynRek(i,F) // i>=0
2     IF F[i]!=0 THEN RETURN F[i];
3     IF i=<1 THEN
4         f=1
5     ELSE
6         f=FibDynRek(i-1,F)+FibDynRek(i-2,F);
```

```
    F[i]=f;
    RETURN f;
```

```
FibDyn(n) // n>=1
    F[]=ALLOC(n); //F_i at F[i-1]
    FOR i=0 TO n-1 DO F[i]=0;
    RETURN FibDynRek(n-1,F);
```

7.2.1 Minimum- Edit- Distance (Levenshtein- Distanz)

Misst Ähnlichkeit von Texten: Wie viele (Buchstaben-)Operationen

- ins(S,i,b) : fügt an i-ter Position Buchstabe b in String S ein
- del(S,i) : löscht an i-ter Position Buchstabe in String S
- sub(S,i,b) : ersetzt an i-ter Position in String S den Buchstaben durch b

sind nötig, um Texte ineinander zu überführen?

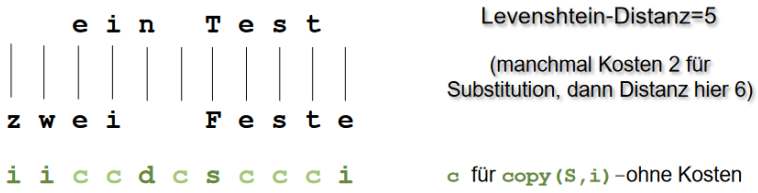


Figure 89: Beispiel

Algorithmische Sichtweise: Überführe schrittweise String $X[1 \dots m]$ von links nach rechts in String $Y[1 \dots n]$. Jeweils Teil $X[1 \dots i]$ bereits in Teil $Y[1 \dots j]$ transformiert.
$D[i][j]$ sei Distanz, um $X[1 \dots i]$ in $Y[1 \dots j]$ zu überführen $(i, j \geq 1)$.
Betrachte nächstens Schritt, um X in Y zu überführen:

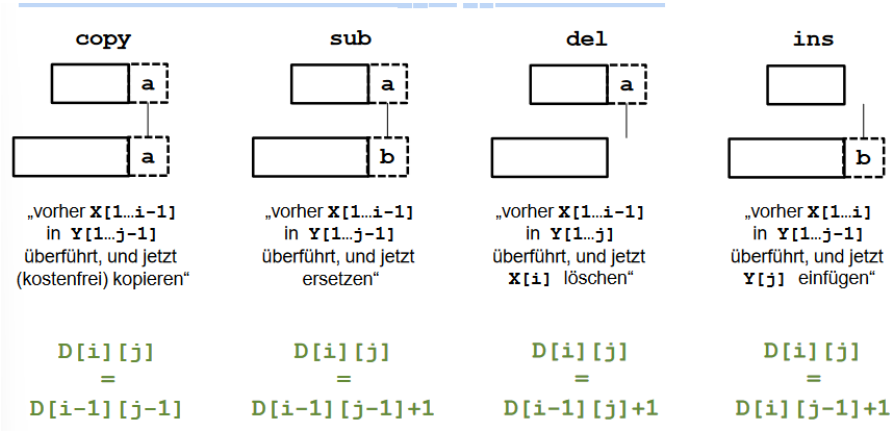


Figure 90: Operationen

Algorithmus mittels dynamischer Programmierung und Memoization:

```
1 MinEditDist(X,Y,m,n) // X=X[1...m], Y=Y[1...n]
2     D[][]=ALLOC(m,n);
3     FOR i=0 TO m DO D[i][0]=i;
4     FOR j=0 TO n DO D[0][j]=j;
5     FOR i=1 TO m DO
6         FOR j=1 TO n DO
7             IF X[i]=Y[j] THEN s=0 ELSE s=1;
8                 D[i][j]=min{D[i-1][j-1]+s,D[i-1][j]+1,D[i][j
                      -1]+1};
9     RETURN D[m][n];
```

Laufzeit und Speicher: $\Theta(mn)$.

7.3 Greedy- Algorithmen

Prinzip: inde Lösung $x = (x_1, x_2, \ldots, x_n)$ indem Teillösung $(x_1, x_2, \ldots, x_{i-1})$ durch Kandidaten x_i ergänzt wird, der lokal am günstigsten erscheint.
Beispiele die wir hatten: Dijkstra, Prim, Kruskal
Funktioniert aber leider nicht immer.

7.3.1 Traveling Salesperson Problem (TSP)

Gegeben vollständiger (un-)gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to R$, finde Tour p mit minimalem Kantengewicht $w(p)$. Eine Tour ist ein Weg $p = (v_0, v_1, \ldots, v_{n-1}, v_n)$ entlang der Kanten $(v_i, v_{i+1}) \in E$ für $i = 0, 1, 2, \ldots, n-1$, der bis auf Start- und Endknoten $v_0 = v_n$ jeden Knoten genau einmal besucht ($V = \{v_0, v_1, \ldots, v_{n-1}\}$).
Graph $G = (V, E)$ ist vollständig, wenn es für alle Knoten $u, v \in V$ mit $u \neq v$

eine Kante $(u, v) \in E$ gibt. Wenn Graph nicht vollständig, aber Tour hat, kann man 'verboten teure' Kanten (u, v) mit $w((u, v)) = |V| \cdot max_{e \in E}\{|w(e)|\} + 1$ hinzufügen.
TSP vs. Dijkstra

- Allgemeiner TSP-Algorithmus: finde optimale Route, die durch jeden Knoten geht und zum Ausgangspunkt zurückkehrt
- Dijsktra-Algorithmus: finde optimalen Pfad vom Ausgangspunkt aus (besucht evtl. nicht alle Knoten und betrachtet auch nicht Rückkehr)

Ansatz Greedy-Algorithmus für TSP: Starte mit beliebigem oder gegebenem Knoten. Nimm vom gegenwärtigen Knoten aus die Kante zu noch nicht besuchtem Knoten, die kleinstes Gewicht hat. Wenn kein Knoten mehr übrig, gehe zu Startpunkt zurück.

```
1 Greedy-TSP(G,s,w) // |V|=n, s starting node
2     FOREACH v in V DO v.color=white;
3     tour[]=ALLOC(n); tour[0]=s; tour[0].color=gray;
4     FOR i=1 TO n-1 DO
5         tour[i]=EXTRACT-MIN(adj(tour[i-1])); //get (white)
              neighbor with minimum edge weight
6         tour[i].color=gray;
7     RETURN tour;
```

Betrachte vollständigen Graphen mit Knoten $V = \{1, 2, 3, \ldots, n\}, n \geq 3$, und folgenden Kantengewichten für beliebig große Konstante N..
$w(\{i, j\}) =$

- $1 : 1 \leq i, i \leq n, j = i + 1$
- $N : i = 1, j = n$
- $2 : sonst$

7.4 Metaheuristiken

Heuristik: dedizierter Suchalgorithmus für Optimierungsproblem, der gute (aber evtl. nicht optimale) Lösung für spezielles Problem findet (problemabhängig, arbeitet direkt 'am' Problem).
Metaheuristik: allgemeine Vorgehensweise, um Suche für beliebige Optimierungsprobleme zu leiten (problem-unabhängig, arbeitet mit abstrakten Problemen).
'Hill- Climb- Strategie:'

```
1 HillClimbing(P) // maxTime constant
2     sol=initialSol(P);
3     time=0;
4     WHILE time<maxTime DO
5         new=perturb(P,sol);
```

Figure 91: Visualisierung

```
6         IF quality(P,new)>quality(P,sol) THEN
7             sol=new;
8         time=time+1;
9     RETURN sol;
```

- *Zeile 2: wählt initiale Lösung*
- *Zeile 5: ändere Lösung leicht ab*
- *Zeile 6: ersetze aktuelle Lösung durch neu, falls diese besser ist*

Beispiel TSP:

- initSol: wähle beliebige Tour, z.B. per Greedy-Algorithmus
- perturb: wähle zwei Knoten u,v zufällig und tausche sie in Tour
- quality: (negatives) Gewicht der aktuellen Tour

Eventuell bleibt Hill-Climbing-Algorithmus in lokalem Maximum hängen, da stets nur leichte Lösungsänderungen in aufsteigender Richtung!
Man kann den Algorithmus vereinfachen, damit der Algorithmus das globale Maximum findet.

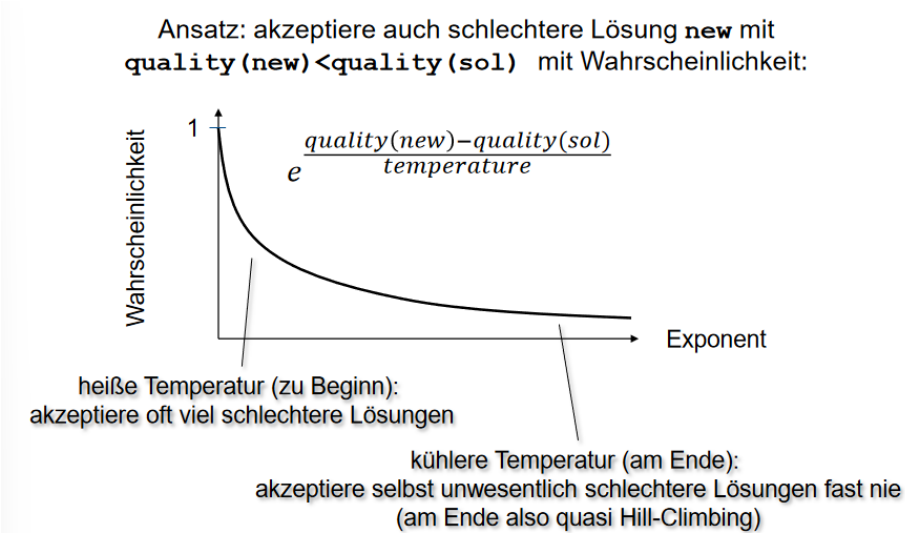


Figure 92: 'In schlechte Richtung' mit Wahrscheinlichkeit

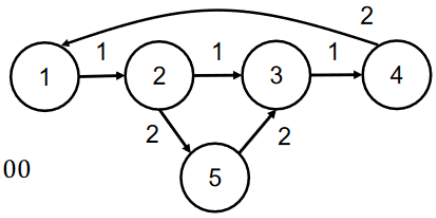
Simulated Annealing:

```
1  SimulatedAnnealing(P) // maxTime constant; TempSched[]
       temparature annealing
2      sol=initialSol(P);
3      time=0;
4      WHILE time<maxTime DO
5          new=perturb(P,sol);
6          temperature=TempSched[time];
7          d=quality(P,new)-quality(P,sol); r=random(0,1);
8          IF d>0 OR r=<exp(d/temperature) THEN
9              sol=new;
10         time=time+1;
11     RETURN sol;
```

Bestimmung eines guten 'Annealing schedule' (Starttemperatur und Abnahme) betrachten wir hier in der Vorlesung nicht weiter.

Starttemperatur
$= \max_{e \in E}\{w(e)\} = N = 1000$

lineare Temperaturabnahme $= N/2n = 100$



	Aktuelle Lösung	Gewicht	Zuf. Tausch	Gew. nach Tausch	Temp	Zuf. Wert r	Exp (-d/temp)	Neue Löung akzeptiert?
1	1→2→3→4→5→1	2N+3	2 und 3	5N	1000	0.37	0.0498	nein
2	1→2→3→4→5→1	2N+3	2 und 4	4N+2	900	0.60	0.1084	nein
3	1→2→3→4→5→1	2N+3	4 und 5	2N+4	800	0.17	0.9986	ja (Zufall)
4	1→2→3→5→4→1	2N+4	2 und 4	5N	700	0.34	0.0138	nein
5	1→2→3→5→4→1	2N+4	3 und 4	3N+3	600	0.45	0.1889	nein
6	1→2→3→5→4→1	2N+4	3 und 5	8	500	0.78	-	ja (besser)
7	1→2→5→3→4→1	8	2 und 5	2N+4	400	0.51	0.0067	nein

Figure 93: Beispiel

Weitere Metaheuristiken:

- Tabu Search: Ausgehend von aktueller Lösung, suche bessere Lösung in der Nähe. Speichere ein Zeit lang schon besuchte Lösungen, und vermeide diese Lösungen. Wenn keine bessere Lösung in der Nähe, akzeptiere auch schlechtere Lösung
- Evolutionäre Algorithmen: Beginne mit Lösungspopulation. Wähle beste Lösungen zur Reproduktion aus. Bilde durch Überkreuzungen und Mutationen der besten Lösungen neue Lösungen. Ersetze schlechteste Lösungen durch diese neue Lösungen
- +Schwarmoptimierung, Ameisenkolonialisierung, ...

8 NP

Ansatz: Problem ist leicht, wenn es in Polynomialzeit lösbar ist. (Worst-Case-) Laufzeit des Algorithmus ist also $\Theta(\sum_{i=0}^{k} a_i n^i) = poly(n)$ (a_i, k konstant).
Halteproblem: Gesucht: Programm H, so dass $H(P) = 1$ falls $P(P)$ anhält - 0 sonst.
Unentscheidbarkeit des Halteproblems: Es gibt kein Programm H, das das Halteproblem löst.

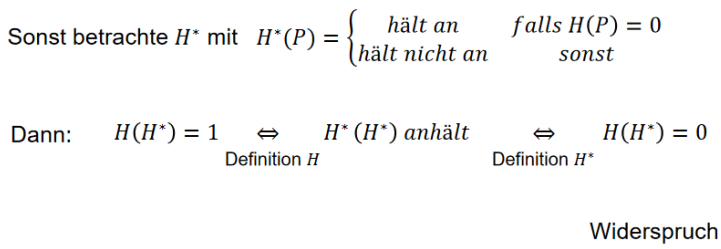


Figure 94: Daraus folgt

8.1 Berechnungsprobleme vs. Entscheidungsprobleme

- Berechnungsproblem:

  Gegeben: Problem P

  Gesucht: Lösung S

  Beispiel:Berechne kürzeste Pfade im Graphen

- Entscheidungsproblem:

  Gegeben: Problem P

  Gesucht: Hat P Eigenschaft E? (0/1-Antwort)

  Beispiel: Ist gerichteter Graph stark zusammenhängend?

Man kann jedes Berechnungs- in ein Entscheidungsproblem überführen, so dass Polynomialzeit-Lösung für Entscheidungsproblem auch Polynomialzeit-Lösung für Berechnungsproblem ergibt.

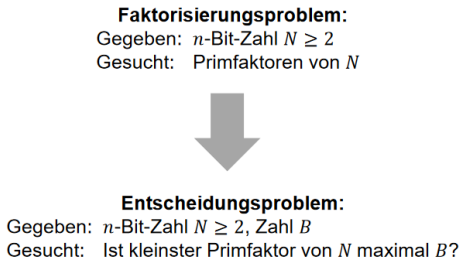


Figure 95: FP zu EP

Beispiel:

```
computeFactor(N) //use decideFactor(N,B) as sub; N>1,
    computes prime factor of N
    L=1; U=N;
    WHILE L!=U DO
        M=L+floor((U-L)/2);
        IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
    RETURN L;
```

```
Factorize(N) // N>1
    WHILE N>1 DO
        p=computeFactor(N);
        print p;
        N=N/p;
```

```
decideFactor(N,B)
    ...
    RETURN d; // d==0 or 1
```

Entscheidungsproblem: Gegeben: n-Bit-Zahl N, Zahl $1 \leq B \leq N$ / Gesucht: Ist kleinster Primfaktor von N maximal B?
In jeder Iteration wird Suchintervall um Hälfte reduziert. Zu Beginn Intervalllänge N, also nach $\Theta(log_2 N) = \Theta(n)$ Iterationen fertig.
Laufzeit: $\Theta(log_2 N) = \Theta(n)$ Iterationen von decideFactor.

Berechnung durch Entscheidung: Sofern Bitlänge der Lösungen poly-

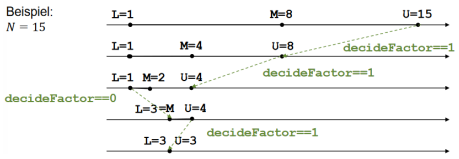


Figure 96: Beispiel

nomiell beschränkt und decide in Polynomialzeit, läuft compute auch in Polynomialzeit.

```
compute(P) //use decide(P,s) as sub
    s=''; // empty string
    IF decide(P,s)==0 THEN RETURN 'no solution';
    done=false;
    WHILE !done DO
        szero=decide(P,s+'0');
        sone =decide(P,s+'1');
        IF szero==0 AND sone==0 THEN
            done=true;
        ELSE IF szero==1 THEN s=s+'0' ELSE s=s+'1';
    RETURN 'solution ' + s;
```

Laufzeit: $\Theta(2 \cdot max_S|S| + 1)$ Iterationen von decide

8.2 Komplexitätsklassen P und NP

Komplexitätsklasse P: Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse P, wenn es einen Polynomialzeit-Algorithmus A_{LE} mit Ausgabe 0/1 gibt, der stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also $P \in L_E \Leftrightarrow A_{LE} = 1$ für alle P gilt.
Beispiel: $L_{SC} = \{G|G$ ist gerichteter, stark zusammenhängender Graph$\}$.
Komplexitätsklasse NP: Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse NP, wenn es einen Polynomialzeit-Algorithmus A_{LE} mit Ausgabe 0/1 gibt, der bei Eingabe eines Zeugen S_P für Eingabe $P \in L_E$ bzw. für jede Eingabe S_P für Eingabe $P \notin L_E$ stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also $P \in L_E \Leftrightarrow \exists S_P : A_{LE}(P, S_P) = 1$ für alle P gilt.

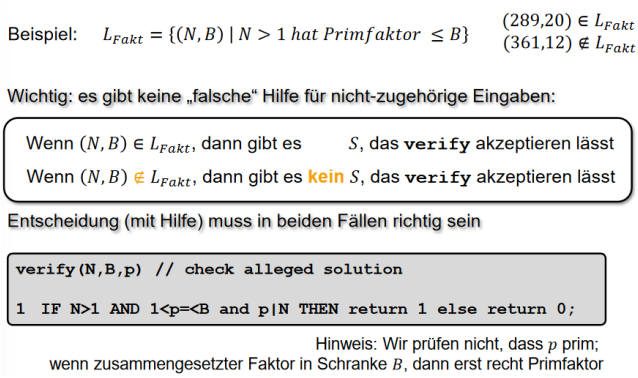


Figure 97: Beispiel

8.2.1 P vs. NP

Jedes Problem in P ist auch in NP: Algorithmus A_{LE} entscheidet ohne Hilfe.
Also: P $\subseteq$ NP , aber bis heute offen, ob auch NP $\subseteq$ P!

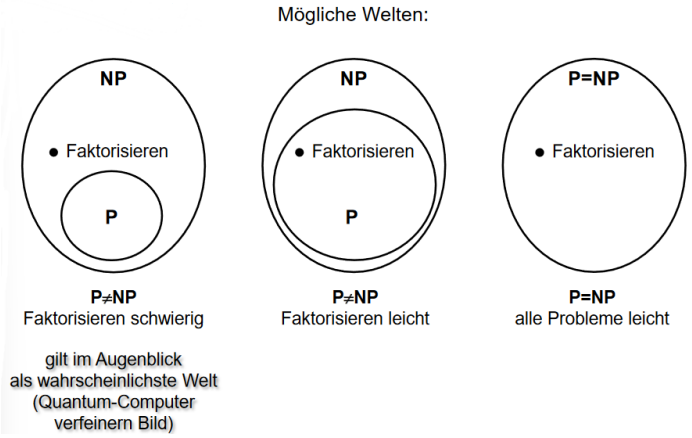


Figure 98: Mögliche Welten

8.3 NP- Vollständigkeit

Ziel: Identifiziere schwierigste Probleme in NP

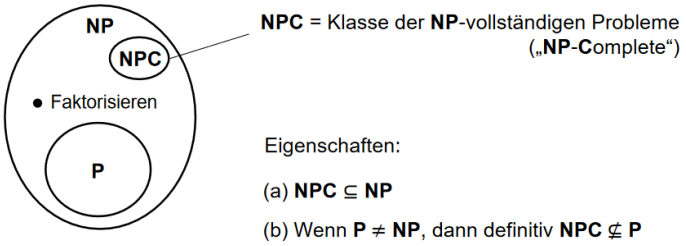


Figure 99: NP- Vollständigkeit

Reduktion: (Reduktion = 'Problemtransformation')
Da Entscheidungsprobleme immer mindestens so schwierig wie Berechnungsprobleme sind, Übertrage auf NP- Entscheidungsprobleme:

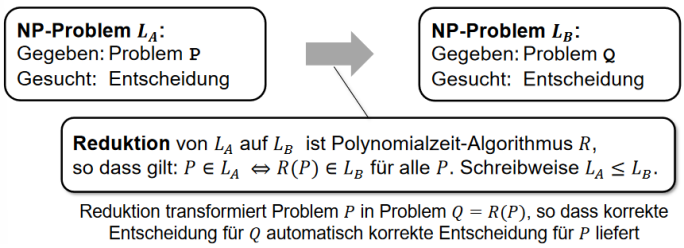


Figure 100: Reduktion

Alle Entscheidungsprobleme L_C aus NP, die mindestens so schwierig wie jedes andere Problem L_A aus NP: $L_A \leq L_C$ für alle $L_A \in$ NP.
Komplexitätsklasse NPC (NP-vollständige Probleme) besteht aus allen Problemen $L_C \in$ NP, so dass $L_A \leq L_C$ für alle $L_A \in$ NP.
Zwei Bedingungen an L_C :

- Problem L_C ist selbst in NP
- jedes NP-Problem darauf reduzierbar ('L_C ist NP-hart')

8.3.1 SAT: Die Mutter aller NP-vollständigen Probleme

SAT:

- Gegeben: Boolesche Formel Φ aus $\wedge, \vee, \neg$ in n Variablen $x_1, x_2, \ldots, x_n$ (Φ polynomielle Komplexität in n)
- Gesucht: Entscheide, ob Φ erfüllende Belegung hat oder nicht
- Beispiel: $\Phi(x_1, x_2, x_3, x_4) = (\neg x_2 \vee (x_3 \wedge \neg x_4)) \wedge x_2 \wedge \neg((x_1 \vee \neg x_2) \wedge x_4)$ hat erfüllende Belegung: $x_1 \leftarrow false, x_2 \leftarrow true, x_3 \leftarrow true, x_4 \leftarrow false$

Offensichtlich: SAT$\in$NP (gegeben Belegung als Zeuge, werte Formel aus)

Reduktion $R(P)$ von L_A auf SAT berechnet: Φ_P(alle Eingabebits) = gültiger Anfangszustand für P $\wedge$ gültige Übergänge $\wedge$ Endzustand mit $d = 1$.

- Wenn P in L_A, gibt es Lösung S, die verifyA akzeptiert mit $d = 1$, dann gibt es aber auch erfüllende Belegung für 'Rechenschritte' Φ_P
- Wenn P nicht in L_A, gibt es keine Lösung S, die verifyA akzeptiert, dann gibt es aber auch keine erfüllende Belegung für 'Rechenschritte' Φ_P

SAT $\leq$ 3SAT:
Boolesche Formeln in konjunktiver Normalform (KNF) mit jeweils 3 Literalen: $\Phi(x_1, x_2, x_3, x_4) = (\neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_3 \vee x_4)$. *(KNF = Und-Verknüpfung von Klauseln; Klausel = Oder-Verküpfng).*

Klausel besteht aus 3 Literalen $X_j \in \{x_j, \neg x_j\}$.
Transformiere, falls weniger Literale in Klausel: $(X_j) = (X_j \vee X_j \vee X_j), (X_j \vee X_k) = (X_j \vee X_k \vee X_k)$.
3SAT:

- Gegeben: Boolesche 3KNF-Formel Φ in n Variablen $x_1, x_2, \ldots, x_n$ (Φ polynomielle Komplexität in n)
- Gesucht: Entscheide, ob Φ erfüllende Belegung hat oder nicht

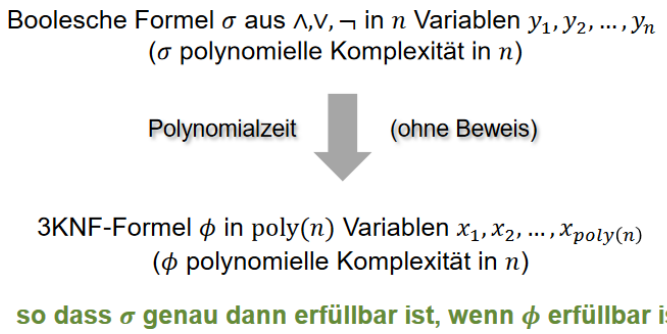


Figure 101: SAT $\Rightarrow$ 3SAT

Folglich: $SAT \leq 3SAT$.
3-Färbbarkeit von Graphen:

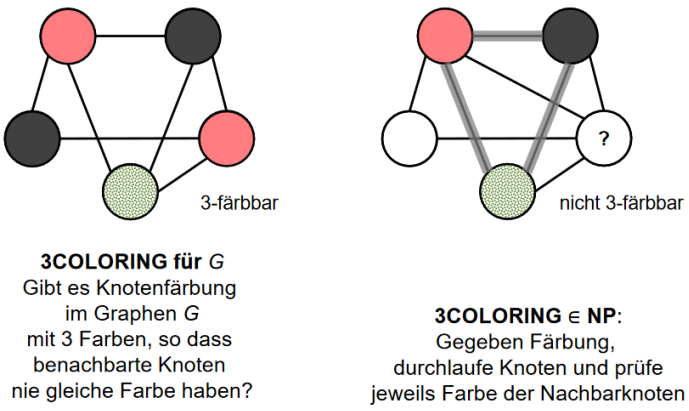


Figure 102: 3- Färbbarkeit Beispiel

Theorem: Wenn Problem L_B NP-vollständig und $L_B \leq L_C$ für $L_C \in$ NP gilt, dann ist auch L_C NP-vollständig.

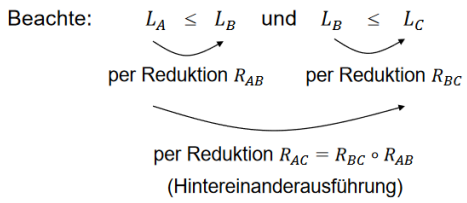


Figure 103:

Also folgt aus 3SAT $\leq$ 3COLORING und 3COLORING $\in$ NP auch, dass 3COLORING NP-vollständig.

NPC- eine Auswahl

- **SAT:** Formel Φ erfüllbar?
- **3SAT:** Formel Φ in 3KNF erfüllbar?
- **3COLORING:** Graph mit drei Farben kantenkonsistent färbbar?
- **HamCycle:** Gibt es Tour im Graphen?
- **TSP:** Gibt es Tour im Graphen, mit Gesamtgewicht $\leq B$?
- **VertexCover:** Gibt es im Graphen Knotenmenge der Größe $\leq B$, so dass jede Kante an einem der Knoten hängt?
- **IndependentSet:** Gibt es im Graphen Knotenmenge der Größe $\leq B$, so dass kein Knotenpaar durch Kante verbunden?
- **Knapsack:** Für Gegenstände mit Wert und Volumen, gibt es Auswahl mit Gesamtwert $\leq W$, aber Gesamtvolumen $\leq V$
- ...

Approximation: NPC-Probleme vermutlich nicht effizient lösbar, aber evtl. leicht approximierbar:

```
3SAT-Approx(Phi, n)
    A[]=ALLOC(n); //assigment for variables
    FOR i=1 TO n DO
        A[i]=true resp. A[i]=false with probability 1/2
    RETURN A;
```

Behauptung: Algorithmus erfüllt im Erwartungswert mind. 1/2 aller Klauseln.
K_i 0-1-Zfúfallsvariable, die angibt, ob i-te Klausel unter Belegung A erfüllt.
Beim m Klauseln folgt aus Linearität des Erwartungswertes: $E[\text{erfüllte Klauseln}] = E[\sum_{i=1}^{m} K_i] = \sum_{i=1}^{m} E[K_i] \geq m/2$

8.4 2-Färbbarkeit und 2SAT in P

Idee: Farbe eines Knoten bestimmt eindeutig Farben der Nachbarknoten.
Ansatz: Beginn mit einem Knoten und beliebiger Farbe. Durchlaufe Graph per BFS, färbe Knoten und identifiziere evtl. Widersprüche.

```
1  2ColoringSub(G,s,col) //G=(V,E), s node
2      s.color=col; newQueue(Q); enqueue(Q,s);
3      WHILE !isEmpty(Q) DO
4          u=dequeue(Q);
5          IF u.color==BLACK THEN nextcol=RED
6          ELSE nextcol=BLACK;
7          FOREACH v in adj(G,u) DO
8              IF v.color==u.color THEN return 0;
9              If v.color==WHITE THEN
10                 v.color=nextcol;
11                 enqueue(Q,v);
12     RETURN 1; //no contradiction
```

Zeile 8 prüft auf Widersprüche; Zeile 9 nimmt nur noch nicht gefärbte Knoten auf
Falls man auf einen Widerspruch trifft, muss man vielleicht nochmal von vorne anfangen.

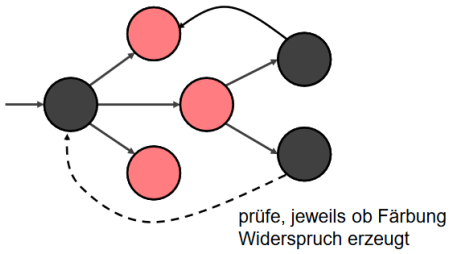


Figure 104: Beispiel, wie ein zwei- färbiger Graph aussieht

Bei ungerichteten Graphen:

```
1  2Coloring(G) // G=(V,E) undirected graph
2      FOREACH u in V Do u.color=WHITE;
3      FOREACH u in V DO
4          IF u.color==WHITE THEN
5              IF 2ColoringSub(G,u,BLACK)==0 THEN return 0;
6      RETURN 1;
```

Laufzeit: $\Theta(|V|+|E|)$
Ändert Lösungsmenge nicht, da verbundene Knoten in beiden Fällen unterschiedliche Farben haben müssen. Bei Neustart keine Kante zwischen Zusam-

menhangskomponenten: Jede individuelle 2-Färbung der Zusammenhangskomponenten kann zu 2-Färbung des Graphen kombiniert werden.

8.4.1 2-SAT

O.b.d.A. stets zwei Literale pro Klausel, sonst schreibe $X_1 = (X_1 \vee X_1)$.
Beispiel: $\Phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$.
Wenn $x_1 \leftarrow false$ gesetzt wird, dann x_2 eindeutig festgelegt für erfüllende Belegung (hier: $x_2 \leftarrow false$), aber: Wenn $x_1 \leftarrow true$ gesetzt wird, dann erstmal noch zwei Möglichkeiten für x_2.
2-SAT $\rightarrow$ Implikationsgraph:

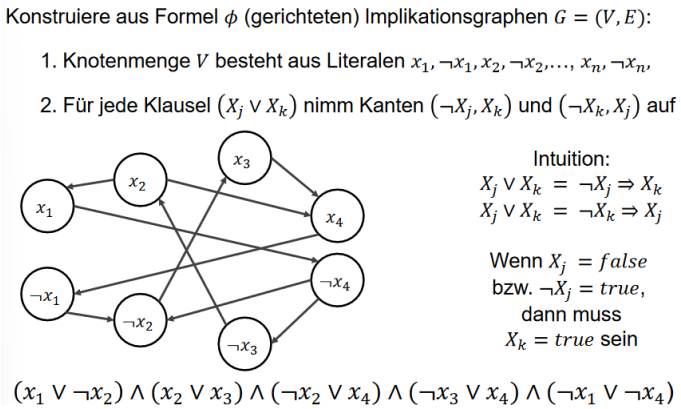


Figure 105: Von 2SAT zum Implikationsgraph

Starke- Zusammenhangskomponente: Formel ist genau dann erfüllbar, wenn in keiner Zusammenhangskomponenten x_j und $\neg x_j$ für ein j liegt.

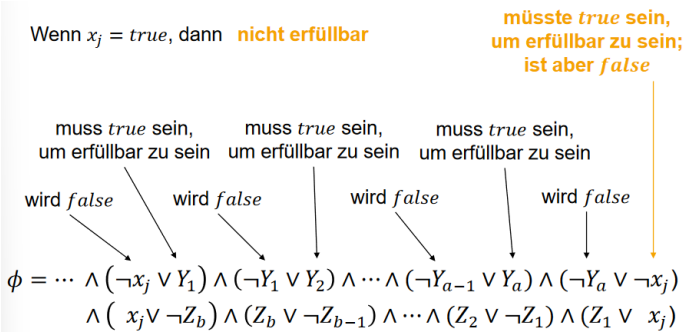


Figure 106: Prüfung Visualisierung

Erfüllende Belegung berechnen: Sortiere 'SCC-dag' topologisch!
'SCC-dag': Graph mit Superknoten aus allen Knoten einer SCC; Kante zwischen SCCs, wenn Kante für zwei Knoten aus SCCs.

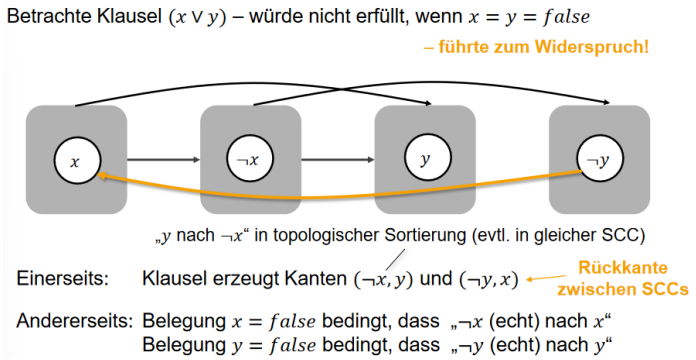


Figure 107: Berechnen

Andere Klausel-Kombinationen $(\neg x \vee y), (x \vee \neg y), (\neg x \vee \neg y)$ führen analog zum Widerspruch. Also werden durch die Belegung alle Klauseln und somit Formel erfüllt.

8.4.2 MAX- 2SAT- Problem

- Gegeben: 2SAT-Formel Φ, Zahl k
- Gesucht: Gibt es Belegung, die mind. k Klauseln erfüllt? $\in$ NPC (ist NP-vollständig)!
- Offensichtlich: MAX-2SAT $\in$ NP. Gegeben Belegung als Zeuge, prüfe, ob mindestens k Klauseln erfüllt werden
- Zeige zusätzlich: 3SAT $\leq$ MAX-2SAT

$$\sigma(x_1, \ldots, x_n) = \cdots \wedge (X_i \vee X_j \vee X_k) \wedge \ldots \qquad (m \text{ Klauseln})$$

eine neue Variable
und 10 Klauseln in ϕ
pro Klausel in σ

$$\begin{gathered}\phi(x_1, \ldots, x_n, w_1, \ldots, w_m) \\ = \cdots \\ \wedge (X_i) \vee (X_j) \wedge (X_k) \wedge (w_h) \\ \wedge \left(\neg X_i \vee \neg X_j\right) \wedge (\neg X_i \vee \neg X_k) \wedge \left(\neg X_j \vee \neg X_k\right) \\ \wedge (X_i \vee \neg w_h) \wedge (X_j \vee \neg w_h) \wedge (X_k \vee \neg w_h) \wedge \ldots\end{gathered}$$

Figure 108: Beispielklausel

Sind mindestens $k = 7m$ Klauseln erfüllbar?
Wenn σ erfüllbar, dann mindestens $k = 7m$ Klauseln in Φ erfüllbar; Wenn σ nicht erfüllbar, dann weniger als $k = 7m$ Klauseln in Φ erfüllbar.

9 *String Matching*

Beim String- Matching geht es darum, dass man eine Sequenz von Buchstaben in einer größeren Sequenz wiederfindet.
Dabei gibt es folgende Regeln:

- T ist der zu durchsuchende Text
- P ist das Textmuster
- lenPat $\leq$ lenTxt
- T, P $\in \Sigma$

Hierfür gibt es 3 Algorithmen:

9.1 *Naiver Algorithmus*

```
1  NaiveStringMatching(T, P)
2      lenTxt = length(T);
3      lenPat = length(P);
4      L = [];
5      FOR sft = 0 TO lenTxt - lenPat DO
6          isValid = true;
7          FOR j = 0 TO lenPat - 1 DO
8              IF P [j] != T[sft + j] THEN
9                  isValid = false;
10         IF isValid THEN
11             L = append(L, sft);
12     RETURN L;
```

Beispiel:
Sei $T = [h, e, h, e, h, h, h, e, y, h]$ und $P = [h, e, h]$.
Daraus ergibt sich:

sft	$T[sft, \ldots, sft + lenPat - 1] \stackrel{?}{=} P$	L
0	**true**	[0]
1	**false**	[0]
2	**true**	[0, 2]
3	**false**	[0, 2]
4	**false**	[0, 2]
5	**false**	[0, 2]
6	**false**	[0, 2]
7	**false**	[0, 2]

Figure 109: Tabelle des Algorithmus

Dieser Algorithmus ist jedoch relativ ineffizient.

9.2 *Endliche Automaten*

Die Idee hierbei ist, dass wir durch einen Endlichen Automaten (FSM) Zwischenspeichern, wie viele Zeichen schon übereinstimmen.
Hierbei ist *st* die Anzahl an schon korrekten Zeichen.

```
FSMMatching(T, delta, lenPat)
    lenTxt = length(T)
    L = []
    st = 0
        FOR sft = 0 TO lenTxt - 1 DO
        st = delta(st, T[sft])
        IF st = lenPat THEN
            L = append(L, sft - lenPat + 1)
    RETURN L
```

Beispiel:
Sei $\Sigma = a, g, n, o$, $T = [g, a, n, a, n, n, a, n, o, n, a, n, a, n, o, g]$ und $P = [n, a, n, o]$.

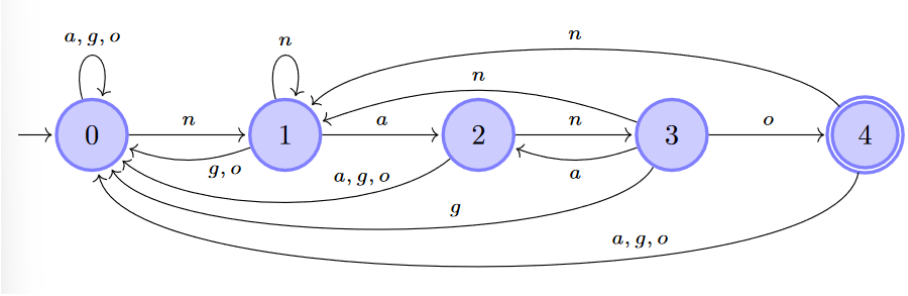


Figure 110: FSM

sft	$T[sft]$	*st*	L	*sft*	$T[sft]$	*st*	L
0	g	0	[]	8	o	4	[5]
1	a	0	[]	9	n	1	[5]
2	n	1	[]	10	a	2	[5]
3	a	2	[]	11	n	3	[5]
4	n	3	[]	12	a	2	[5]
5	n	1	[]	13	n	3	[5]
6	a	2	[]	14	o	4	[5, 11]
7	n	3	[]	15	g	0	[5, 11]

Figure 111: Tabelle des Algorithmus

9.3 *Rabin-Karp Algorithmus*

Der Rabin-Karp-Algorithmus basiert auf folgender Überlegung:
Für jedes $0 \leq sft \leq lenTxt - lenPat$, sei t_{sft} der Dezimalwert des Teilstrings $T[sft, v..., sft + lenPat - 1]$, und sei p der Dezimalwert von P. Offenbar ist sft eine gültige Verschiebung, also $T[sft, ..., sft + lenPat - 1] = P$, genau dann wenn $t_{sft} = p$.

```
RabinKarpMatchBasic(T, P)
    n = T.length;
    m = P.length;
    h = 10^(m-1);
    p = 0;
    t0 = 0;
    L = [];
    FOR i = 0 TO m - 1 DO
        p = (10p + P [i]);
        t0 = (10t0 + T [i]);
    FOR sft = 0 TO n - m DO
        IF p == tsft THEN
            L = append(L, sft);
        IF sft < n - m THEN
            t_(sft+1) = 10(t_(sft) - T[sft]h) + T[sft + m];
    RETURN L
```

```
RabinKarpMatch(T, P, q)
    n = T.length;
    m = P.length
    h = 10^(m-1) (mod q)
    p = 0;
    t0 = 0;
    L = [];
    FOR i = 0 TO m - 1 DO
        p = (10p + P[i]) (mod q)
        t0 = (10t0 + T[i]) (mod q)
    FOR sft = 0 TO n - m DO
        IF p == tsft THEN0
            b = true;
            FOR j = 0 TO m - 1 DO
                IF P[j] != T[sft + j] THEN
                    b = false;
                    break;
            IF b THEN
                L = append(L, sft);
        IF sft < n - m THEN
            t_(sft+1) = (10(t_(sft) - T [sft]h) + T[sft + m
                ]) (mod q)
    RETURN L
```

Beispiel:
Sei $P = [7, 3, 4]$, $T = [6, 9, 1, 7, 3, 4, 5, 0, 9, 4, 6, 2, 4, 8, 7, 3, 4]$ und $q = 13$.
Hieraus folgt p=6 (mod 13)

sft	$t_{sft} \pmod q$	$t_{sft} == p \pmod q$	$T[sft, \ldots, sft + lenPat - 1] == P$	Treffer?
0	2	false		
1	7	false		
2	4	false		
3	6	**true**	**true**	Ja
4	7	false		
5	8	false		
6	2	false		
7	3	false		
8	10	false		
9	7	false		
10	0	false		
11	1	false		
12	6	**true**	false	Unecht
13	2	false		
14	6	**true**	**true**	Ja

Figure 112: Tabelle des Algorithmus

Der Algorithmus findet also die Verschiebung $L = [3, 14]$.