Merkblatt zu Algorithmen und Datenstrukturen

Tobias Kratz

Marius Glock Vu Do Le

Merkblatt für Algorithmen und Datenstrukturen im Sommersemester 2019 der TU Darmstadt

Inhaltsverzeichnis

Allgemein Algorithmen

Probleme der Informatik

Ein **Problem** in der Informatik enthält eine Beschreibung der Ein- Ausgabe aber keinen Übergang. *Ein Beispiel wäre, sortiere die Menge an Wörtern*

Eine **Probleminstanz** ist ein spezifiziertes Problem mit konkreter Belegung der Eingabe, in der eine Entsprechende Ausgabe erwünscht ist. *Bsp.: Berechne x=23*

> »Ein Algorithmus ist eine **endliche Sequenz** von Handlungsvorschriften, die eine **Eingabe** in eine **Ausgabe** transformiert.«[1]

Aufgaben und Eigenschaften eines Algorithmus

Ein Algorithmus soll

- Endlich: Die Notation hat eine endliche Länge
- Eindeutig: Jeder Einzelschritt ist klar definiert und ausführbar. Die Reihenfolge der Einzelschritte ist festgelegt.
- Spezifizierung der Ein- Ausgabe: Anzahl und Typen aller Elemente ist definiert.

Weiterhin kann ein Algorithmus folgende Eigenschaften haben

- Determiniertheit: Für die gleiche Eingabe immer die gleiche Ausgabe berechnet
- Determinismus: Für die gleiche Eingabe ist die Ausführung und die die Ausgabe stets identisch
- Terminierung: Der Algorithmus läuft für jede Eingabe nur endlich lange
- Korrektheit: Der Algorithmus berechnet stets die spezifizierte Ausgabe (falls diese terminiert)
- Effizienz: Sparsamkeit im Ressourcenverbrauch (Zeit, Speicher...)

Effizienz von Algorithmen

- Effizienzfaktoren
  - Rechenzeit (Anzahl der Einzelschritte)
  - Kommunikationsaufwand (z.B. Netzwerk)
  - Speicherbedarf
  - Zugriff auf Speicher (z.B. Festplatte)
- Laufzeit hängt von verschiedenen Faktoren ab
  - Länge der Eingabe
  - Implementierung der Basisoperationen
  - Takt der CPU

Sortieralgorithmen

Das Sortierproblem

Zu Sortierende Elemente auch Schlüsselwerte.
Bedingung zum Sortieren: Schlüsselwerte müssen verglichbar sein.
Eingabe eine Sequenz von Schlüsselwerten <a1, a2, ..., an>
Ausgabe : Permutation <a1', a2', ..., an'> mit z.B. der Eigentschaft a1' <= a2' <= a3' <= .... <= an'
Eingabe ist eine **Instanz** des Sortierproblems.
Algorithmus ist **korrekt**, wenn dieser das Problem für alle Instanzen löst.

Vergleichbarkeiten von Algorithmen

- Berechnungsaufwand: O(n)
- Effizienz: Best Chase vs. Worst Chase vs. Average Case
- Speicherbedarf:
  - in-place: zusätzlicher Speicher, von der Eingabegröße Unabhängig,
  - out-of-place: Speicherbedarf von der Eingabegröße abhängig

[1] Corm et al.,2009

- Stabilität: stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht, (wichtig bei mehrfachen Sortierung nach verschiedenen Schlüsselwerten)

Korrektheit

- Schleifeninvarianten: Zu Beginn jeder Iteration der for-Schleife besteht die Teilfolge A[0..j-1] aus den Elementen der ursprünglichen Teilfolge A[0...j-1] enthaltenen Elementen, allerdings in sortierter Reihenfolge.
- Schleifeninvariante muss 3 Dinge erfüllen
  - Initialisierung Invariante ist vor jeder Iteration wahr.
  - Fortsetzung Wenn die Invariante vor jeder Schleife wahr ist, dann bleibt sei auch bis zum Beginn der nächsten Iteration wahr.
  - Terminierung Wenn die Schleife abbricht, liefert uns die Invariante eine Eigenschaft, die uns hilft zu zeigen, dass der Algorithmus korrekt ist.

Laufzeiten

Komplexität. Abstrakte Rechenzeit T(n) ist abhängig von den Eingabedaten

Problem	T(n)
Sortieren einer n-elementigen Menge	Vertauschungen und Vergleiche
Suchen in einer n-elementigen Menge	Vergleiche zu durchlaufende Knoten
Auswertung einer rekursiven Funktion	Funktionsaufrufe

Asymptotische Notationen. Verhalten von T(n) für große $n \in \mathbb{N}$. Komplexität ist unabhängig von konstanten Faktoren und Summanden.
Berücksichtigen nicht in Betrachtung

- Rechnergeschwindigkeit
- Aufwände der Initialisierung

Komplexitätsmessungen via Funktionsklassen

Θ-Notation. Funktionen $f, g : \mathbb{N} \to \mathbb{R}_{>0}$
$\mathbb{N}$ = Eingabelänge
$\mathbb{R}$ = Zeit
$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c_2 g(n)\}$ wobei:

- f = Funktion f
- $\exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ = Positive Konstanten und $\forall n \geq n_0$
- $0 \leq f(n) \leq c_2 g(n)$ $f(n)$ wird von $c_1 g(n)$ und $c_2 g(n)$ für hinreichend große n eingeschlossen

Schreibweise: $f \in \Theta(g)$, manchmal auch $f = \Theta(g)$.
Θ-Notation beschränkt eine Funktion asymptotisch von **oben** und **unten**

***O*-Notation.** Obere asymptotische Schranke
$O(g) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$ wobei:

- f = Funktion f
- $\exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ = Positive Konstanten und $\forall n \geq n_0$
- $0 \leq f(n) \leq cg(n)$: f(n) wird von cg(n) für hinreichend große n beschränkt

Sprechweise: f wächst höchstens so schnell wie O(g)
Schreibweise: $f = O(g)$

Ω-Notation. Untere asymptotische Schranke.
$\Omega(g) = f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)$ wobei:

- f = Funktion f
- $\exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ = Positive Konstanten
- $\forall n \geq n_0$ Für alle n größer gleich n_0
- $0 \leq cg(n) \leq f(n) = f(n)$ wird von $cg(n)$ für hinreichend große n beschränkt

Schreibweise: $f = \Omega(g)$
Mit allen 3 Notationen folgt: Für zwei beliebige $f(n)$ und $g(n)$ gilt $f(n) = \Theta(g(n))$ genau dann, wenn die Gleichungen $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind.

Komplexitätsklassen. n ist die Länge der Eingabe

Klasse	Kosten	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Expoentiell	
$\Theta(n!)$	Faktoriell	Permutationen

o-Notation und ω-Notation. nicht asymptotische scharfe Schranken
$o(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0, 0 \leq f(n) < cg(n)\}$
Gilt für **alle** Konstanten c>0. In O-Notation gilt es für eine Konstante c>0
$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$

Divide-and-Conquer

Idee Zerlege das Problem und löse es direkt oder durch weitere Zerlegung.

Divide Teile das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des gleichen Problems darstellen.

Conquer Beherrsche die Teilprobleme rekursiv. Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme auf direktem Weg.

Combine Vereinige die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems.

Für ein kleines Problem $n \leq c$ (c = Konstante), dann braucht direkte Lösung $\Theta(1)$. Sonst gilt: Aufteilen in je a Teilproblme der Größe $\frac{1}{b}$ des Ursprünglichen Problems. Für das lösen für a Teilprobleme der Größe $\frac{n}{b}$ ist $aT(\frac{n}{b})$. $D(n)$ ist die Zeit, die die Aufteilung braucht und C(n) die Zeit zum zusammenführen. Ist nicht $n \leq c$, so ist die Laufzeit $aT(\frac{n}{b}) + D(n) + C(n)$.

Lösung von Rekusionsgleichungen

Substitutionsmethode. Eine Schranke erraten und mit math. Induktion erraten.

Rekursionsbaum-Methode. Jeder Knoten stellt ein Teilproblem da. Die Wurzel stellen die analysierenden Kosten dar, die Blätter die Basiskosten. Berechnen die Kosten innerhalb jeder Ebene des Baums. Die Gesamtkosten ist die Summe über die Kosten aller Eben. Die Methode nutzt eine Lösung zu raten die man dann mit der Substitutionsmethode überprüfen kann.

Mastermethode. liefert Schranken für Rekursionsgleichungen der Form $T(n) = aT(\frac{n}{b}) + f(n)$ mit $a \geq 1.b > 1$ und $f(n)$ eine asymptotisch positive Funktion. in je a Teilproblme der Größe $\frac{1}{b}$ des Ursprünglichen Problems. Das lösen dauert für a Teilprobleme der Größe $\frac{n}{b}$ und $f(n)$ stellt dann die gesamten Kosten zum lösen dar.

Mastertheorem. $a \geq 1$ und $b > 1$ sind Konstanten. Sei f(n) eine positive Funktion und T(n) für alle ≥ 0 $T(n) = aT(\frac{n}{b}) + f(n)$, obwohl wir $\frac{n}{b}$ so interpretieren, dass damit entweder $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ gemeint ist, Dann hat T(n) die folgenden Schranken:

- Gilt $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.
- Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} lgn)$.
- Gilt $f(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und $af(\frac{n}{b}) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend großen n, dann ist $T(n) = \Theta(f(n))$.

Das Mastertheorem deckt 3 Fälle nicht ab, bei deren Eintreten das Mastertheorem nicht angewendet werden kann.

- Wenn $f(n)$ kleiner ist als $n^{log_b a}$ aber nicht polynomial kleiner
- Wenn $f(n)$ größer ist als $n^{log_b a}$ aber nicht polynomial größer
- Regularitätsbedingung $af(\frac{n}{b}) \leq cf(n)$ wird nicht erfüllt.

Insertion Sort

Kurzbeschreibung. Es wird ein key initialisiert, der dann mit den Array Einträgen von dem key verglichen wird. Solange der Wert im Array vor dem key (A[1]) größer ist, als, der key, wird der Eintrag um eins nach rechts geschoben, und der Index um eins verkleinert. Ist der key dann an der richtigen Stelle, wird er an der i+1sten Stelle im Array eingefügt. Dies wird solange wiederholt, bis das Array einmal durchlaufen ist (bis A.length-1).
inkrementelle Herangehensweise.

Pseudo Code. Pseudo Code des Insertion Sort Algorithmus

```
Array Insertion-Sort(A){
1 for j=1 TO A.length-1
2     key = A[j]
3     i=j-1
4     while i>=0 && A[i]>key
5         A[i+1]=A[i]
6         i=i-1
7     A[i+1]=key
```

Korrektheit von Insertion Sort.

- Initialisierung Beginn mit j=1, Teilfeld A[0 ... j-1] besteht nur aus A[0], dieses Element ist sortiert.
- Fortsetzung Die For-Schleife sorgt dafür, dass die Elemente A[j-1],A[j-2],... um je eine Stelle nach rechts verschoben werden, bis A[j] an der korrekten Stelle steht. Das Teilfeld A[0..j] besteht aus ursprünglichen Elementen und ist sortiert. Die Inkrementation von j erhält die Invariante.
- Terminierung Abbruchbedingung for-Schleife, wenn *j>A.length-1*. Jede Iteration erhöht j. Dann bei Abbruch ist *j=n* und einsetzten in Invariante liefert das Teilfeld *A[0...n-1]* welches aus den ursprünglichen Elementen besteht und sortiert ist. Teilfeld ist gesamtes Feld.

Der Algorithmus arbeitet korrekt

Zeile	Kosten	Anzahl
1	c_1	n
2	c_2	n-1
3	0	n-1
4	c_4	n-1
5	c_5	$\sum_{j=1}^{n-1} t_j$
6	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$
7	c_7	$\sum_{j=1}^{n-1} (t_j - 1)$
8	c_8	n-1

Laufzeit. T(n) ist die Summe aus allen Zeilen Kosten, also

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=1}^{n-1} t_j +$$

$$c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n-1)$$

- Best Case Das Feld ist bereits sortiert. Dann gilt $t_j = 1$ für $j = 1, ..., n-1$

  $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$

  $$(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

  Die Laufzeit ist eine lineare Funktion in n.

- Worst Case Das Feld ist in umgekehrter Reihenfolge sortiert. Die While-Schleife geht über alle Elemente A[0... j-1]. Somit gilt $t_j = j + 1$ für $j = 1, ..., n-1$. Die Laufzeit ist eine quadratische Funktion in n

Bubble Sort

Kurzbeschreibung. Vergleiche Paare von benachbarten Schlüsselwerten und tausche das Paar, falls rechter Schlüsselwert kleiner ist als linker.

Pseudo Code. Pseudo Code des Bubble Sort Algorithmus

```
BUBBLE–SORT(A)
1  for i=0 TO A.length-2
2     for j=A.length-1 DOWNTO i+1
3           if A[j]<A[j-1]
4                 SWAP(A[j],A[j-1])
```

Laufzeit. Worst Case und Best Case Laufzeiten sind identisch, da stets alle Elemente der Teilfolge miteinander verglichen werden, Anzahl der Vertauschungen sind im Best case 0 und im Worst case $\frac{n^2-n}{2}$

Merge Sort

Kurzbeschreibung. Teile die Folge in n Elementen in zwei Teilfolgen von je $\frac{n}{2}$ Elementen auf (Divide). Sortiere dann die zwei Teilfolgen rekursiv (Conquer) und mische die zwei sortierten Teilfolgen, um die sortierte Lösung zu erzeugen (Combine).

Pseudo Code. Pseudo Code des Merge Sort Algorithmus

```
MERGE(A, p, q, r)
1    n1 = q - p + 1
2    n2 = r -q
3    new array L[o...n1] && R[0...n2]
4    for i=0 to n1 - 1{
5         L[i] = A[p +1 i] }
6    for j=0 to n2 - 1{
7         R[j] = A[q + j + 1] }
8    L[n1] = ∞
9    R[n2] = ∞
10   i = 0
11   j = 0
12   for k = p to r
13        if L[i] <= R[j]
14             A[k] = L[i]
15             i = i + 1
16        else A[k] = R[j]
17             j = j + 1
```

Laufzeit. Da wir höchstens n Grundschritte für das Vermischen ausführen, liegt die benötigte Zeit bei $\Theta(n)$

Korrektheit.

- **Schleifeninvariante** Zu Beginn jeder Iteration der for-Schleife (Zeile 12-17) enthält das Teilfeld A[p..k-1] die k - p kleinsten Elemente aus L[0..n1] B und R[0..n2] in sortierter Reihenfolge. Weiter sind L[i] und R[j] die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.

- **Initialisierung** Vor der ersten Iteration gilt k = p. Daher ist A[p..k1] leer und enthält 0 kleinste Elemente von L und R. Wegen i = j = 0 sind L[i] und R[j] die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurde.

- **Fortsetzung** Müssen zeigen, dass Schleifeninvariante erhalten bleibt. Dafür nehmen wir an, dass L[i] ≤ R[j] . Dann ist L[i] kleinstes Element, welches noch nicht zurück kopiert wurde. Da Array A[p..k1] die kp kleinsten Elemente enthält, wird der Array A[p..k] die kp+1 kleinsten Elemente enthalten nachdem der Wert nach der Durchführung von Zeile 14 kopiert wurde. Nach Erhöhung der Variablen k und i stellt die Schleifeninvariante für die nächste Iteration wieder her. Wenn L[i] > R[j] dann analoges Argument mit Zeilen 16-17.

- **Terminierung** Beim Abbruch gilt k=r+1. Durch die Schleifeninvariante enthält A[p. . r] die kleinsten Elemente von L[0...n1] und R[0..n2] in sortierter Reihenfolge. Alle Elemente außer der Sentinals wurden korrekt zurück kopiert.

Analyse. Vereinfachende Annahme: Größe des Problems ist eine Potenz von 2. Ziel: Bestimmte Rekursionsgleichung für Laufzeit T(n) von n Zahlen im schlechtesten Fall. Mitte des Feldes wird berechnet, dies benötigt konstant $\Theta(1)$ (Devide). Dann werden rekursiv zwei Teilprobleme der Größe $\frac{n}{2}$ gelöst (Laufzeit $2T(\frac{n}{2})$(Conquer). Die Merge Prozedur auf ein Teilfeld der Länge n mit linearer Zeit, also $\Theta(n)$. Die Laufzeit T(n) ist ist $\Theta(1)$, falls n = 1 oder $2T(\frac{n}{2})+\Theta(n)$, falls n > 1. Löst man die Rekursionsgleichung, beträgt die Laufzeit

$$\Theta(nlgn)$$

Quicksort

Kurzbeschreibung. Ein Pivotelement x wählen und das Array in zwei Teilarrays aufteilen. Das erste Element enthält alle $y \leq x$, das zweite alle $y > x$. Die Teillisten rekursiv mit Quicksort sortieren. Die Einelementigen Listen sind schon sortiert.
Das Array A[p..r] in A[p...q-1] und A[q+1...r] zerlegen. Den Index von p als Teil von Partition berechnen (Divide). Die beiden Teilarrays rekursiv mit Quicksort sortieren (Conquer). Da die Teilarrays bereits sortiert sind, ist A[p...r] schon beim zusammenführen sortiert (Combine).

Pseudo Code. Der Pseudo Code des Algorithmus

```
QUICKSORT(A,p,r)
1    if p<r
2        q = PARTION(A,p,r)
3        QUICKSORT(A,p, q 1 )
4        QUICKSORT(A,q+1,r)
```

und der Partition Code

```
PARTITION(A,p,r)
1    x=A[r]
2    i=p-1
3    for j=p to r-1
4        if A[j] <= x
5            i=i+1
6            swap(A[i] & A[j])
7    swap(A[i+1],A[r])
8    return i+1
```

Laufzeit. Laufzeit von PARTION auf dem Teilarray beträgt $\Theta(n)$ mit n=rp+1. Die Laufzeit hängt von der balance der Teilarrays ab. Sind sie balanciert, so läuft der Quicksort so schnell wie Merge Sort, sind sie es nicht, ist er so langsam wie Insertion-Sort. Insgesamt gilt im Worst Case $T(n) = \Theta(n^2)$. Im Best Case kostet die Zerlegung = $(nlgn)$

Korrektheit. Zu Beginn jeder Iteration der for-Schelife gillt für Arrayindex k 1. Ist $p \leq k \leq i$, so gilt $A[k] \leq x$. 2. Ist $i+1 \leq k \leq j-1$, so gilt $A[k] > x$. 3. Ist $k = r$, so gilt $A[k] = x$.

- **Initialisierung:** Vor der ersten Iteration gilt i=p-1 und j=p. Da keine Werte zwischen p und j liegen, liegen auch keine zwischen i+1 und j-1, somit sind die ersten beiden Eigenschaften erfüllt. Die Zuweisung in Zeile 1 erklärt die 3.
- **Fortsetzung:** Durch Zeile 4 ergibt sich entweder A[j]>x, dann inkrementiert die Schleife nur den Index j. Dann gilt 2. und alle anderen Einträge bleiben unverändert. Wenn A[j]≤, dann wird Index inkrementiert und die Einträge A[i] und A[j] getauscht und der Index j erhöht. Wegen des Vertauschens gilt $A[i] \leq x$ und 1. ist erfüllt. Analog gilt A[j-1]>x, da das Element welches mit A[j-1] vertauscht wurde wegen der Invariante größer als x ist.
- **Terminierung:** Es gilt j=r. Daher gilt, dass jeder Eintrag des Arrays zu einer der drei durch die Invariante beschriebenen Menge gehört.

Randomized QS. Will man Quicksort zufällig laufen lassen, gilt

```
RANDOMIZED-PARTION(A,p,r)
1    i = RANDOM(p,r)
2    SWAP(A r, A[i])
3    RETURN PARTION(A, p, r)

RANDOMIZED-QUICKSORT(A,p,r)
1    if p<r
2        q = RANDOMIZED-PARTION(A,p,r)
3        RANDOMIZED-QUICKSORT(A,p, q 1 )
4        RANDOMIZED-QUICKSORT(A,q+1,r)
```

Grundlegende Datenstrukturen

Abstrakte Datentypen und Datenstrukturen

Die Übergang der Begriffe ist fließend, Abstrakte Datenstrukturen (ADTs) werden oft als Datenstruktur bezeichnet.
Abstrakter Datentyp ist näher an der Anwendung und beschreibt **was** realisiert werden soll.
Bsp.: Stack mit Operationen isEmpty, pop, push
Datenstrukturen ist näher an der Maschine und besagt **wie** es realisiert wird.
Bsp.: Stack-Operationen als Array oder verkettete Liste

Stack

Stacks werden nach dem LIFO Prinzip realisiert

Laufzeiten. Push $\Theta(1)$| Pop $\Theta(1)$

Realisierungen.

- **Als Array:** Annahme: maximale Größe das der Stack haben darf ist bekannt
  Prinzip: Der Stack besitzt einen Pointer als Attribut der immer auf das oberste Element zeigt. Wird dem Stack etwas gepusht oder gepopt zeigt es auf das neue Element bzw. das Element das noch auf dem Stack liegt.
  Optimierungsideen: Wird die Annahme verletzt, also das Array wird überfüllt, muss in ein größeres Array überführt werden oder auf viele andere Arrays verteilt werden(verkette Listen) Wenn in ein größeres Array überführt wird, kostet dies Zeit, da man keinen unendlichen Speicher hat. Es muss also entschieden werden wie groß das Array werden soll mit einer **RESIZE** Methode, der Standard ist das Doppelte der jetzigen Größe.
  Methoden:

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

pop(S)
1    if isEmpty(S) THEN
2        error      underflow
3    else
4        S.top=S.top-1;
5    return S.A[S.top+1];

isEmpty(S)
1    if S.top<0 THEN
2        return true
3    else
4        returnfalse;

push(S,x)
1    if S.top==MAX-1
2        error overflow
3    else
4        S.top=S.top+1;
5        S.A[S.top]=x;

Mit RESIZE(A,m)

new(S)
1    S.A[]=ALLOCATE(1);
2    S.top=-1;
3    S.memsize=1;

pop(S)
1    if isEmpty(S)
2        error underflow
3    else
4        S.top=S.top-1;
5        if 4*(S.top+1)==S.memsize
6            S.memsize=S.memsize/2;
7            RESIZE(S.A[],S.memsize);
8        return S.A[S.top+1];

push(S,x)
1    S.top=S.top+1;
2    S.A[S.top]=x;
3    if S.top+1>=S.memsize THEN
4        S.memsize=2*S.memsize;
5        RESIZE(S.A[],S.memsize);
```

- **Als verkettete Liste:** Prinzip: Der Head ist hierbei der Pointer. Doppelt verkette Listen werden benutzt. Jeder Eintrag besteht aus dem Wert(key), dem Vorgänger(prev) und Nachfolger(next).Die verkettete Liste kann auch auf einem Array realisiert werden. Implementierung mittels Wächter zur Vereinfachung (NIL). Beim Löschen darauf achten das Vorgänger des zu löschendem Element auch wieder verknüpft wird, sowie der Head richtig gesetzt wird. Gleiches beim Einfügen mit dem neuem Element.

Verkette Listen

Verkettete Listen sind Elemente die untereinander einen Nachfolger und Vorgänger haben. Sie haben immer einen Head, welcher den Startpunkt darstellen soll und manchmal einen Tail, welches das Ende der Kette darstellt. Vereinfachung durch Wächter(NIL) Achte beim Löschen und Einfügen immer auf die Nachfolger und Vorgänger.

Laufzeiten. Einfügen $\Theta(1)$| Löschen $\Theta(1)$ | Suchen $\Theta(n)$
Laufzeit der Löschung eines Wertes ist $\Omega(n)$

Realisierungen.

- **Als Elementobjekte:** Grundauf Implementierung wobei die Objekte auf ihren Wert(key), Vorgänger und Nachfolger als Attribut besitzen. Die eigentliche Liste besitzt in der Regel nur einen Verweis auf den Head also dem Startelement.
  Methoden:

```
search(L,k)
//returns pointer to k in L (o rnil)
1    current=L.head;
2    while current!=nil & current.key!=k
3        current=current.next;
4    return current;

insert(L,x)
//inserts element x in L
1    x.next=L.head;
```

```
2    x.prev=nil;
3    if L.head!= nilTHEN
4        L.head.prev=x;
5    L.head=x;

delete(L,x)
//deletes element x from L
1    if x.prev!= nilTHEN
2        x.prev.next=x.next
3    else
4        L.head=x.next;
5    if x.next!= nilTHEN
6        x.next.prev=x.prev;

Vereinfachung mit W chter
deleteSent(L,x)
// deletes x from L with sentinel
1    x.prev.next=x.next;
2    x.next.prev=x.prev;
```

- **Als Array:**
  1. Zeile Arrayposition
  2. Zeile Wertzahlen
  3. Zeile Darstellung das es immer key, prev, next ist
  L.head ist auf 6 gesetzt

0	1	2	3	4	5	6	7	8
12	6	nil	-	-	-	45	nil	0
key	prev	next	key	prev	next	key	prev	next

Abstrakter Datentyp Queue

Arbeitet nach dem FIFO Prinzip. Elemente die eingefügt(enqueue) werden kommen nach hinten und nur das Element ganz vorne kann entfernt werden(dequeue). Um dies zu realisieren muss man den Anfang und das Ende der Struktur kennen (Q.front und Q.end).

Laufzeiten. Enqueue $\Theta(1)$| Dequeue $\Theta(1)$

Realisierungen.

- **Als "virtuelles"zyklisches Array:**
  Problematisierung: Die Grenzen des Arrays(Anfang und Ende) könnten erreicht werden, deswegen wird es als zyklisches Array dargestellt.
  Prinzip: Wenn man am Ende des Arrays ist, springt man wieder zum Anfang und vice versa. Q.front wandert zyklisch nach rechts(Index erhöht sich außer beim Anfang/Ende) beim dequeue und Q.rear beim enqueue.
  Optimierungsidee: Dabei gibt es aber ein Problem, da es keine Unterscheidung gibt ob die Schlange leer ist oder voll, weswegen wir den Zustand abspeichern müssen oder ein Element als Abstandhalter verwenden.
  Methoden:

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

isEmpty(Q)
1        return Q.empty;

dequeue(Q)
1    if isEmpty(Q) THEN
2        e r r o r underflow
3    else
4        Q.front=Q.front+1 modMAX;
5    if Q.front==Q.rearTHEN
6        Q.empty=true;
7    returnQ.A[Q.front-1 modMAX];

enqueue(Q,x)
1    if Q.rear==Q.front & !Q.empty
2        error      overflow
3    else
4        Q.A[Q.rear]=x;
5        Q.rear=Q.rear+1 modMAX;
6        Q.empty=false;
```

- **Als einfach verkette Liste:**
  Prinzip: Q.front dient jetzt als head und wandert nach rechts beim dequeue und Q.rear beim enqueue.
  Methoden:

```
new(Q)
1    Q.front=nil;
2    Q.rear=nil;

isEmpty(Q)
1    if Q.front== nil
2        return true;
3    else
4        return false;

dequeue(Q)
1    if isEmpty(Q)
2        error "underflow"
3    else
4    x=Q.front;
5    Q.front=Q.front.next;
6    return x;

enqueue(Q,x)
```

```
1    if isEmpty(Q)
2        Q.front=x;
3    else
4        Q.rear.next=x;
5    x.next=nil;
6    Q.rear=x;
```

Binär Bäume

Binär Bäume besitzen Knoten, diese Zeigten auf 0 bis 2 Nachfolger-Knoten, diese sind somit die Kinder-Knoten. Einen Knoten ohne Nachfolger, nennt man Blatt. Eine Knoten mit nur einem Nachfolger ist ein Halbblatt. Den Knoten, der am höchsten im Baum steht nennt man Wurzel. Jeder Knoten im Baum kann als Wurzel seines eigenen Teilbaumes gesehen werden.

Realisierung.

- **Als verkettet Liste** Jeder Knoten erhält einen Wert(key), sowie eine Array von Zeigern auf die Kinder(child[]). Zusätzlich kann ein Zeiger auf den Elternkonten verwendet werden(parent). Für nicht-leern Baum gibt es #Knoten-1 viele Einträge ≠ **nil** über alle Listen child[].
- **Als (ungerichteter) Graph**

Inorder-Traversieren. Inorder-Traversieren gibt einen Binär Baum in aufsteigender Reihenfolge aus.

- ```
  INORDER(x)
  1 if x != nil then
  2   INORDER(x.left)
  3   print x.key
  4   INORDER(x.right)
  ```
- **Laufzeit** Behauptung: $T(n) = O(n)$. Für $T(0) = c$ bei leerem Baum. Über Rekursion mit k Knoten im linken Teilbaum und $n-k-1$ im rechten:

$$T(n) = T(k) + T(n-k-1) + d$$

$$\leq (c+d)k + c + (c+d)(n-k-1) + c + d$$

$$\leq (c+d)n + c$$

Pre- und Postorder-Traversieren.

- **Preorder** Preorder für Kopien. Preorder betrachtet zunächst alle Knoten und legt Kopien an und geht dann in die Teilbäume und kopiert diese

```
Preorder(x)
1 if x != nil then
2   print x.key
3   preorder(x.left)
4   preorder(x.right)
```

- **Postorder** Postorder für Löschung. Postorder geht erst in die Teilbäume und löscht diese. Dannach erst betrachtet Postorder die Knoten und löscht diesen. Zeiger auf rechten Teilbaum noch vorhanden, wenn linker bereits gelöscht wurde.

```
Preorder(x)
1 if x != nil then
2   preorder(x.left)
3   preorder(x.right)
4   print x.key
```

Durch Pre-, Post- oder Inorder lässt sich ein Binär Baum nicht eindeutig beschreiben. Dies lässt sich durch Pre- oder Postorder und Inorder gemeinsam realisieren. Dabei lässt sich durch Pre- oder Postorder jeweils die Wurzel identifizieren und durch Inorder jeweils die Werte der linken und rechten Teilbäume.

Suche.

- ```
  search(x,k)
  1 if x==nil then return nil;
  2 if x.key=k then return x;
  3 L=search(x.left, k);
  4 if L != nil then return L;
  5 return search(x.right,k);
  ```

  Wobei x den aktuellen Knoten und k den gesuchten Wert beschreibt.
- **Laufzeit** Die Laufzeit liegt in $\Theta(n)$, da jeder Knoten maximal einmal besucht wird, dies gilt auf für den Worst Case

Einfügen.

- ```
  insert(T,x)
  //x.parent==x.left==x.right==nil;
  1 if T.root != nil THEN
  2 T.root.parent=x;
  3 x.left=T.root;
  4 T.root=x;
  ```

  Wobei T den Baum und x den neuen Wert beschreibt.
  Achtung: Erzeugt linkslastigen Baum!!11
- **Laufzeit** Die Laufzeit ist konstant und somit in $\Theta(1)$

Löschen

- **Idee:** Ersetze X durch Halbblatt ganz rechts. **Sonderfälle**: Halbblatt ist selbst x oder die Wurzel.
- ```
  connect(T,y,w)
  //connects w to y.parent
  1 v=y.parent;
  2 if y != T.root THEN
  ```

```
3    if y == v.right THEN
4        v.right=w;
5    else
6        v.left=w;
7  else
8    T.root=w;
9  if w != nil THEN
10   w.parent=v;
```

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

- **Laufzeit** Die Laufzeit wird dominiert von der Suche nach dem rechtesten Knoten, dieser wird nach maximal h(Höhe des Baumes, h=n möglich) Schritten erreicht, das Verpflanzen des Knotens ist dabei konstant und die Laufzeit für das löschen liegt in $\Theta(h)$

Binäre Suchbäume

Sie sind ein Unterart der Binär Bäume, sie besitzen eine totale Ordnung auf den Werten. Jeder Knoten-Wert im linken Teilbuam ist $\leq$ dem Knoten und jeder Knoten rechts ist $\geq$ dem Knoten. In einem BST(Binary Search Tree) genügt die Pre- oder Postorder um einen Baum eindeutig zu beschreiben. Die Inorder beschreibt den Baum hingegen nicht eindeutig.

- **Suche**

```
search(x,k) //1.Aufruf x=root
1  if x==nil OR x.key==k THEN
2    return x;
3  if x.key > k THEN
4    return search(x.left,k)
5  else
6    return search(x.right,k);
```

Mit einer Laufzeit von $O(h)$, da mit jedem besuchten Knoten die Richtung zum gesuchten Knoten gezeigt wird, entsteht ein eindeutiger Pfad und es müssen nicht alle Knoten besucht werden.

- **Einfügen**

```
insert(T,z)
//may insert z again
//z.left==z.right==nil;
1  x=T.root; px=nil;
2  while x != nil DO
3    px=x;
4    if x.key > z.key THEN
5        x=x.left
6    else
7        x=x.right;
8  z.parent=px;
9  if px==nil THEN
10   T.root=z
11 else
12   if px.key > z.key THEN
13       px.left=z
14   else
15       px.right=z;
```

Das einfügen eines neuen Knotens wird dominiert vom Suchen des geeigneten Platzes und wird somit wieder durch $O(h)$ beschrieben.

- **Löschen**

```
transplant(T,u,v)
1  if u.parent==nil THEN
2    T.root=v
3  else
4    if u==u.parent.left THEN
5        u.parent.left=v
6    else
7        u.parent.right=v;
8  if v != nil THEN
9    v.parent=u.parent;
```

```
delete(T,z)
1  if z.left==nil THEN
2    transplant(T,z,z.right)
3  else
4    id z.right==nil THEN
5        transplant(T,z,z.left)
6    else
7        y=z.right;
8        while y.left != nil
             y=y.left;
9        if y.parent != z THEN
10           transplant(T,y,y.right);
11           y.right=z.right;
12           y.right.parent=y;
13       transplant(T,z,y);
14       y.left=z.left;
15       y.left.parent=y;
```

Die Laufzeit für das Transplantieren ist konstant und liegt in $\Theta(1)$ die Laufzeit der Löschen-Methode wird somit wieder von der Suche nach dem Knoten dominiert und ist somit beschränkt durch $O(h)$.

Erweiterte Datenstrukturen

Rot-Schwarz-Bäume

Eigenschaften.

- Jeder Knoten hat die Farbe rot oder schwarz
- Die Wurzel ist schwarz, so fern 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“).

Schwarzhöhe. eines Knotens x ist die Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt oder Halbblatt im Teilbaum des Knotens. Ein Rot-Schwarz-Baum mit n Knoten hat maximale Höhe $h \leq 2 \cdot \log_2(n+1)$

Einfügen. 1. Finde Elternknoten y wie im BST 2. Färbe neuen Knoten z rot 3. Stelle RS-Baum-Bedingungen wieder her.

```
rotateLeft(T,x)
1    y=x.right;
2    x.right=y.left;
3    if y.left != nil
4         y.left.parent=x;
5    y.parent=x.parent;
6    if x.parent==T.sent
7         T.root=y
8    else
9         if x==x.parent.left
10             x.parent.left=y
11        else
12             x.parent.right=y;
13   y.left=x;
14   x.parent=y;
```

Die Laufzeit beträgt $\Theta(1)$

```
insert(T,z)
//z.left==z.right==nil;

1    x=T.root; px=T.sent;
2    while x != nil DO
3         px=x;
4         if x.key > z.key
5              x=x.left
6         else
7              x=x.right;
8    z.parent=px;
9    if px==T.sent
10        T.root=z
11   else
12        if px.key > z.key
13             px.left=z
14        else
15             px.right=z;
16   z.color=red;
17   fixColorsAfterInsertion(T,z);

fixColorsAfterInsertion(T,z)
1    while z.p.color==red DO
2         if z.p=z.p.p.left
3              y=z.p.p.right;
4              if y.color==red
5                   z.p.color=black;
6                   y.color=black;
7                   z.p.p.color=red;
8                   z=z.p.p;
9              else
10                  if z==z.p.right
11                       z=z.p;
12                       rotateLeft(T,z);
13                  z.p.color=black;
14                  z.p.p.color=red;
15                  rotateRight(T,z.p.p);
16        else
17             //exchange left and right
18   T.root.color=black;
```

Laufzeiten. Die Laufzeit beträgt $O(h) = O(logn)$. Das ist auch die Laufzeit für das löschen. Im Worst Case sind die Laufzeiten für Einfügen, Löschen und Suchen $\Theta(logn)$

```
delete(TreeNode toDelete) {
1    TreeNode x = _nil;
2    TreeNode y = toDelete;
3    TreeNode.NodeColor y_original_color = y.color;
4    if( toDelete.left == _nil ) {
5        x = toDelete.right;
6        transplant(toDelete, toDelete.right);
7    } else if( toDelete.right == _nil ) {
8        x = toDelete.left;
9        transplant(toDelete, toDelete.left);
10   } else {
11       y = minimumSubtree(toDelete.right);
12       y_original_color = y.color;
13       x = y.right;
14       if( y.p == toDelete ) {
15                x.p = y;
16       } else {
17                transplant(y, y.right);
18                y.right = toDelete.right;
19                y.right.p = y;
```

```
20        }
21        transplant(toDelete, y);
22        y.left = toDelete.left;
23        y.left.p = y;
24        y.color = toDelete.color;
25    }
26    if(y_original_color == BLACK) {
27        delete_fixup(x);
28        }
29    }

delete_fixup(TreeNode x) {
1  while( x!=_root && x.color == BLACK ) {
2   if( x == x.p.left ) {
3   TreeNode w = x.p.right;
4    if( w.color == RED ) {
5     w.color = BLACK;
6     x.p.color = RED;
7     left_rotate(x.p);
8     w = x.p.right;
9    }
10   if( w.left.color == BLACK
    && w.right.color == BLACK){
11    w.color = RED;
12    x = x.p;
13   } else {
14    if( w.right.color == BLACK ) {
16     w.left.color = BLACK;
17     w.color = RED;
18     right_rotate(w);
19     w = x.p.right;
20    }
21    w.color = x.p.color;
22    x.p.color = BLACK;
23    w.right.color = BLACK;
24    left_rotate(x.p);
25    x = _root;
26    }
27   } else {
28    TreeNode w = x.p.left;
29    if( w.color == RED){
30     w.color = BLACK;
31     x.p.color = RED;
32     right_rotate(x.p);
33     w = x.p.left;
34    }
35    if( w.right.color == BLACK
 && w.left.color == BLACK ) {
36     w.color = RED;
37     x = x.p;
38    } else {
39     if( w.left.color == BLACK ) {
40       w.right.color == BLACK;
41       w.color = RED;
420      left_rotate(w);
43       w = x.p.left;
44    }
45    w.color = x.p.color;
46    x.p.color = BLACK;
47    w.left.color = BLACK;
48    right_rotate(x.p);
49    x = _root;
50   }
51 }
52 }
53 x.color = TreeNode.NodeColor.BLACK;
54 }
```

AVL-Bäume

Jeder Knoten im Baum erfüllt die "Balanceregel": B(x)= Höhe(rechter Teilbaum) - Höhe (linker Teilbaum), wobei x der Knoten ist B(x) deren Balance Dabei dürfen die Teilbäume nur eine maximale Differenz von 1 haben also $B(x) \in 1, 0, +1$. Ein AVL Baum mit n Knoten hat maximale Höhe $h \leq 1,441 \cdot \log_2 n$. Im Vergleich zu RS-Bäumen verletzten AVL Bäume häufiger die Baum-Bedingungen, was zu mehr Aufwand führt. Da die Baumhöhe jedoch geringer ist, ist die Suche schneller.

> Jeder AVL-Baum der Höhe h lässt sich als Rot-Schwarz Baum mit Schwarzhöhe $\lceil \frac{h}{2} \rceil$ darstellen. Allgemeiner: Für ungerades h gibt es sogar einen Baum mit roter Wurzel für Schwarzhöhe $\frac{h-1}{2}$, der alle anderen RS-Baumbedingungen erfüllt. Achtung: Für jede Höhe $h \geq 4$ gibt es einen RSB, der kein AVL-Baum ist.

Laufzeiten. Einfügen, Löschen, Suchen $\Theta(1)$

Einfügen. Fall 1 und 3) Sei "x"der Knoten bei dem eine B(x) Differenz von +-2 besteht und der eingefügte Knoten "k"an den äußersten Enkel-Teilbaum. Wenn "k"am rechtestem Enkel-Teilbaum ist führe Methode rotateLeft(T,x) an Knoten T aus. Wenn am linkestem dann rotateRight(T,x).
Fall 2 und 4) Dieser Fall deckt die Enkel-Teilbäume dazwischen ab. Sei nun x der Knoten bei dem eine B(x) Differenz von +-2 ist, "y"einer der Kindknoten von x und "z"einer der Enkelknoten in der innerem von x, bei dem am Teilbaum ein neuer Knoten "k"angefügt wurde. Wenn z rechts ist wird zuerst an y rotateRight(T,y) ausgeführt und danach erst an x mit rotateRight(T,x).Vice verca wenn z links ist.

```
insert(T,z)
//z.left==z.right==nil;

1    x=T.root; px=T.sent;
2    while x != nil DO
```

```
3       px=x;
4       if x.key > z.key THEN
5           x=x.left
6       else
7           x=x.right;
8   z.parent=px;
9   if px==T.sent THEN
10      T.root=z
11  else
12      if px.key > z.key THEN
13          px.left=z
14      else
15          px.right=z;
16  fixBalanceAfterInsertion(T,z);
```

Die Laufzeit ist $O(h) = O(logn)$

Löschen

. 1) y is left child of z and x is left child of y (Left Left Case) do Right Rotate (z)
2) y is left child of z and x is right child of y (Left Right Case) do Left Rotate (y), Right Rotate(z)
3) y is right child of z and x is right child of y (Right Right Case) do Left Rotate(z)
4) y is right child of z and x is left child of y (Right Left Case) Right Rotate (y), Left Rotate(z)
Die Fälle muss man dann bis hin zur Root überprüfen.

[2]

Rebalance.

Splay-Bäume

Splay-Bäume sind selbst organisierende Datenstrukturen. Man geht davon aus das neu eingefügte oder gesuchte Knoten nach oben öfters angefragt werden, deswegen werden Sie an die Wurzel gespült. Sei k zum Beispiel ein neuer Knoten wird dieser durch Rotationen (Zick, Zack und ihre Kombinationen) zum Root. Bei der Amortisierte Laufzeitanalyse, die die Laufzeit pro Operation über mehrere Operationen hinweg betrachtet, ist :

> Für m ≥ n Operationen auf einem Splay-Baum mit maximal n Knoten ist die Worst-Case-Laufzeit $O(m \cdot \log_2 n)$ also $O(m \cdot \log_2 n)$ Operation

Splay-Methode. Die unteren Methoden sind eine Zusammenfassung der Kombinationen von Zick und Zack z.B. zig-zig enthält auch zag-zag Rotationen

```
splay(T,z)
1   WHILE z != T.root DO
2   IF z.parent.parent==nil THEN
3       zig(T,z);
4   ELSE
5       IF z==z.parent.parent.left.left
        OR
        z==z.parent.parent.right.right THEN
6           zigZig(T,z);
7       ELSE
8           zigZag(T,z);

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

/*Falls x das linke Kind seines Vaters
ist und keinen Gro vater hat,
und somit bereits direkt unter der Wurzel
steht,wird eine
zick-Rotation (Rechts-Rotation)
durchgefuehrt. Nun ist x die neue Wurzel
des Baumes und die Splay-Operation
beendet. Liegt x  im rechten
Teilbaum seines Vaters,
wird analog eine zack-Rotation
(Links-Rotation) durchgef hrt. */
zig(T,z)
1   IF z==z.parent.left THEN
2       rotateRight(T,z.parent);
3   ELSE
4       rotateLeft(T,z.parent);

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

Suchen. Im Grunde wie beim binären Suchbaum, jedoch muss das gesuchte Element wieder hochgespült werden. Die Suche an sich kostet O(h), der Splay kostet auch O(h), insgesamt also O(h+h) bzw. O(h).

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

[2] https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

```
10        splay(T,x);
11        return T.root;
```

Einfügen. 1. Suche analog zum Einfügen bei BST Einfügepunkt. Kostet O(h)
2. Spüle vermeintlichen Elternknoten y von x per Splay-Operation nach oben. Kostet O(h)
3. Splitte Baum auf. Kostet O(1)

> Wenn y ≤ x ist, splitte Kante von y und rechtem Teilbaum und vice-versa mit y ≥ x

4. Baue Baum mit neuem Knoten als Wurzel zusammen
Wenn y ≤ x füge den abgetrennten rechten Teilbaum an der rechte Seite von x an und y an der linken Seite. Vice-versa mit y ≥ x. Kostet O(1)
Insgesamt kostet es O(h) Zeit.

Löschen. 1. Spüle gesuchten Knoten x per Splay-Operation nach oben. Kostet O(h)
2. Lösche x, wenn einer der Teilbäume leer ist, sind wir fertig. Kostet O(1)
3. Spüle den „größten" Knoten y in L per Splay-Operation nach oben, y kann kein rechtes Kind haben, da es der größter Wert in L ist. Kostet O(h+h) bzw. Kostet O(h)
4. Hänge R an y an. Kostet O(1)
Insgesamt kostet es O(h) Zeit.

Heaps

Achtung: Heaps sind keine BSTs, linke Kinder können größere Werte haben als die rechten Kinder. Heaps können für den Abstrakten Datentyp **Priority Queue** Anwendung finden.

(Binäre Max-) Heaps. Ist ein binärer Baum, der bis auf das unterste Level vollständig und im untersten Level von links nach rechts gefüllt wird. Für alle Knoten **x** ≠ **T.root** gilt: **x.parent.key** ≥ **x.key**. Da der Baum (fast) vollständig ist, gilt h ≤ ($\log n$)+1.

Heaps als Array. H.size speichert die Anzahl der Konten. Duale Sichtweise als Pointer oder als Array:

- j.parent = $\lceil \frac{j}{2} \rceil - 1$
- j.left = $2(j+1) - 1$
- j.right = $2(j+1) - 1$
- Blätterindizes sind zwischen: $\lceil \frac{(n-1)}{2} \rceil$ und n-1

Einfügen. Idee: An letzter Stelle einfügen und dann nach oben im Baum auf die richtiger Stelle tauschen. Algorithmus beispielhaft für Max-Heap:

```
insert(H,k) //as unlimited Array
1 H.size=H.size+1;
2 H.A[H.size-1]=k;
3 i=H.size-1;
4 while i>0 && H.A[i] > H.A[i.parent]
5    SWAP(H.A,i,i.parent);
6    i=i.parent;
```

Der neue Wert muss im Worts Case an die Wurzel und der Algorithmus muss auf jeder Ebene eine Tauschaktion durchführen. Die Laufzeit ist also $O(h) = O(\log n)$

Wurzel Löschen. Idee: Der Wurzel Knoten (Max/Min) wird ausgegeben und entfernt. Die Wurzel wird durch das letzte Blatt ersetzt und dieses wird solange mit seinem Kind getauscht bis die Heap-Eigenschaft wieder gegeben ist. Algorithmus beispielhaft für Max-Heap:

```
extract-max(H) //as unlimited Array
1 if isEmpty(H) THEN
2    return error        underflow
3 else
4    max=H.A[0];
5    H.A[0]=H.A[H.size-1];
6    H.size=H.size-1;
7    heapify(H,0);
8    return max;
```

Algorithmus beispielhaft für Max-Heap:

```
heapify(H,i) //as unlimited Array
1 maxind=i;
2 if i.left<H.size &&
     H.A[i]<H.A[i.left] THEN
3    maxind=i.left;
4 if i.right<H.size &&
     H.A[maxind]<H.A[i.right] THEN
5    maxind=i.right;
6 if maxind != i THEN
7    SWAP(H.A,i,maxind);
8    heapify(H,maxind);
```

Die Laufzeit dieses Algorithmuses wird von heapify(H,i) dominiert, da der Knoten im Worts Case zu einem Blatt werden muss. Die Laufzeit beträgt $O(h) = O(\log n)$

Heap-Konstruktion. Idee: Das Array ist bereits kopiert und nun wird heapify für alle Knoten deren Index ≤ $\lceil \frac{(n-1)}{2} \rceil - 1$ (Keine Blätter) aufgerufen.

```
buildHeap(H.A)
//Array A schon nach H.A kopiert
1 H.size=A.length;
2 for i = ⌈(H.size-1)/2⌉ - 1 DOWNTO 0 DO
3    heapify(H.A,i);
```

Der Algorithmus besitzt eine Laufzeit von $O(n \cdot h) = O(n \cdot \log n)$

Heap-Sort. Idee: Die Werte eines Arrays werden zu einem Heap.Die Werte des Heaps werden dann nach und nach entnommen und ausgegeben. Der Algorithmus beispielhaft für Max-Heaps:

```
heapSort(H.A) //Array A schon nach H.A kopiert
1 buildHeap(H.A);
```

```
2 WHILE !isEmpty(H) DO
3     extract-max(H);
```

Die Laufzeit liegt ebenfalls in $O(n \cdot h) = O(n \cdot \log n)$

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 (keys) hat. Die Wurzel besitzt zwischen 1 und $2t-1$ Key
- Die Werte innerhalb eines Knoten aufsteigend geordnet sind
- die Blätter alle die gleiche Höhe haben
- jeder innere Knoten mit n Werten $n+1$ Kinder hat, so dass für alle Werte k_j aus dem j-ten Kind gilt: $k_1 \leq key[1] \leq k_2 \leq key[2] \leq k_n \leq key[n] \leq k_{n+1}$
- Anzahl Werte n im B-Baum im Vergleich zur Höhe h $n \geq 1+(t-1)\cdot\sum_{i=1}^{h-1}(2t^{i-1}) = 1+2(t-1)\cdot\frac{t^{h-1}-1}{t-1} = 2t^{h-1}-1$ also: $\log_t \frac{n+1}{2} + 1 \geq h$
- Je größer t desto "flacher"ist der B-Baum
-

B-Bäume finden in MySQL Datenbanken Anwendung, die meisten Indices werden dort in B-Bäumen gespeichert.

Darstellung.

- x.n - Anzahl der Werten eines Knoten x
- x.key[0],...,x.key[x.n-1] -(geordnere) Werte in Knoten x
- x.child[0],...,x.child[x.n] -Zeiger auf Kinder in Knoten x

Suche. Idee: Suche funktioniert ähnlich, wie die iterative Suche auf einem BST, mit der Ausnahme, dass in den Konten zusätzlich horizontal gesucht wird.

```
search(x,k)
1 while x != nil DO
2     i=0;
3     while i < x.n &&
x.key[i] < k DO i=i+1;
4     if i < x.n && x.key[i]==k THEN
5         return (x,i);
6     else
7         x=x.child[i];
8 return nil;
```

Die Laufzeit ergibt sich aus der While-Schleife (Zeile 1) und die While-Schleife (Zeile 3). Zeile 3 hat im Worst Case 2t Iterationen und somit eine Laufzeit die durch $O(1)$ geschränkt ist. Somit wird die Laufzeit des Alogrithumeses durch $O(t \cdot h) = O(\log_t n)$ beschränkt.

Einfügen. Idee: Ein neuer Wert wird immer in ein Blatt eingefügt. Wenn das Blatt weniger als $2t-1$ Werte besitzt, einfach einfügen, sonst mittleren Wert in Elternkonten einfügen, sollte Elternknoten mehr als $2t-1$ Werte besitzen, dann rekursiv nach oben. Split an der Wurzel erzeugt neue Wurzel.

```
BTree insert(K key, V value) {
1 if (mRoot == null) {
2   mRoot = createNode();
3 }
4 ++mSize;
5 if (mRoot.mCurrentKeyNum ==
 BTNode.UPPER_BOUND_KEYNUM) {
6   // The root is full, split it
7   BTNode<K, V> btNode = createNode();
8   btNode.mIsLeaf = false;
9  btNode.mChildren[0] = mRoot;
10  mRoot = btNode;
11  splitNode(mRoot, 0,
btNode.mChildren[0]);
12 }
13 insertKeyAtNode(mRoot, key, value);
14 return this;
16}
```

```
insertKeyAtNode(BTNode rootNode, K key, V value)
1 {
2 int i;
3 int currentKeyNum = rootNode.mCurrentKeyNum;
4 if (rootNode.mIsLeaf) {
5   if (rootNode.mCurrentKeyNum == 0) {
6       // Empty root
7       rootNode.mKeys[0] =
new BTKeyValue<K, V>(key, value);
8       ++(rootNode.mCurrentKeyNum);
9       return;
10  }
11 // Verify if the specified key
doesn't exist in the node
12 for (i = 0; i < rootNode.mCurrentKeyNum; ++i)
{
13  if (key.compareTo(rootNode.mKeys[i].mKey)
== 0) {
14      // Find existing key, overwrite its
value only
15      rootNode.mKeys[i].mValue = value;
16      --mSize;
17      return;
18  }
19 }
20 i = currentKeyNum - 1;
21 BTKeyValue<K, V> existingKeyVal =
rootNode.mKeys[i];
22 while ((i > -1) &&
```

```
(key.compareTo(existingKeyVal.mKey)
< 0)) {
23   rootNode.mKeys[i + 1] =
existingKeyVal;
24   --i;
25   if (i > -1) {
26       existingKeyVal =
rootNode.mKeys[i];
27   }
28 }
29 i = i + 1;
30 rootNode.mKeys[i] =
new BTKeyValue<K, V>(key, value);
31 ++(rootNode.mCurrentKeyNum);
32 return;
33 }

splitNode(BTNode parentNode,
int nodeIdx, BTNode btNode) {
1 int i;

2 BTNode<K, V> newNode = createNode();
3 newNode.mIsLeaf = btNode.mIsLeaf;

4 // Since the node is full,
5 // new node must share
LOWER_BOUND_KEYNUM
(aka t - 1) keys from the node
6 newNode.mCurrentKeyNum =
BTNode.LOWER_BOUND_KEYNUM;

7 // Copy right half of the keys
from the
node to the new node
8 for (i = 0;
i < BTNode.LOWER_BOUND_KEYNUM; ++i) {
9    newNode.mKeys[i] =
btNode.mKeys[i + BTNode.MIN_DEGREE];
10   btNode.mKeys[i +
BTNode.MIN_DEGREE] = null;
11 }

12 // If the node is an internal node
(not a leaf),
13 // copy the its child pointers
at the half right as well
14 if (!btNode.mIsLeaf) {
15   for (i = 0; i < BTNode.MIN_DEGREE;
++i) {
16       newNode.mChildren[i] =
btNode.mChildren[i +
BTNode.MIN_DEGREE];
17       btNode.mChildren[i +
BTNode.MIN_DEGREE] =
null;
18   }
19 }


20 // The node at this point should
have LOWER_BOUND_KEYNUM
(aka min degree - 1) keys at
this point.
21 // We will move its right-most key
to its parent node later.
22 btNode.mCurrentKeyNum =
BTNode.LOWER_BOUND_KEYNUM;

23 // Do the right shift for r
elevant child pointers of the
parent node
24 // so that we can put the new
node as its new child pointer
25 for (i = parentNode.mCurrentKeyNum;
i > nodeIdx; --i) {
26   parentNode.mChildren[i + 1] =
parentNode.mChildren[i];
27   parentNode.mChildren[i] = null;
28 }
29 parentNode.mChildren[nodeIdx + 1] =
newNode;

30 // Do the right shift all the keys
of the parent node the right side of
the node index as well
31 // so that we will have a slot for
move a median key from the
splitted node
32 for (i =
parentNode.mCurrentKeyNum - 1;
i >= nodeIdx; --i) {
33   parentNode.mKeys[i + 1] =
parentNode.mKeys[i];
34   parentNode.mKeys[i] = null;
35 }
36 parentNode.mKeys[nodeIdx] =
btNode.mKeys[
BTNode.LOWER_BOUND_KEYNUM];
37 btNode.mKeys[
BTNode.LOWER_BOUND_KEYNUM] =
null;
38 ++(parentNode.mCurrentKeyNum);
}
```

Das Einfügen eines Wertes kann eine maximale Laufzeit von $O(t \cdot h) = O(\log_t n)$ haben.

Löschen. Idee: Liegt der Wert in einem Blatt, so kann dieser Wert einfach entfernt werden, sollte das Blatt darauf-

hin "zu leer"sein, so wird rotiert oder verschmolzen. Wird ein innerer Konten gelöscht, so wird verschoben oder verschmolzen.
Informeller Algorithmus:

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

Zu Zeile 3: Aktueller Knoten hat zu diesem Zeitpunkt mindestens t Werte, sonst wäre er vorher verschmolzen worden oder es wäre rotiert worden. Beim Verschmelzen/Verschieben des Kindes kann die Anzahl der Werte im aktuellen Knoten nicht unter t-1 fallen.
Zu Zeile 4: Entfernen aus Blatt problemlos, da mindestens t Werte. Entfernen im inneren Knoten durch Verschieben oder Verschmelzen.
Der Algorithmus besitzt eine Laufzeitgrenze von $O(t \cdot h) = O(\log_t n)$

Zusammenfassung. B-Bäume besitze für jede Operation eine Laufzeit von $\Theta(\log_t n)$

Achtung:
Die O-Notation versteckt (konstanten) Faktor t für die Suche innerhalb eines Konten; $t \cdot \log_t n = t \cdot \frac{\log_2 n}{\log_2 t}$ ist in der Regel größer als $\log_2 n$, also in der Regel nur Vorteilhaft, wenn Daten blockweise eingelesen werden.

Zufällige Datenstrukturen

Bisher war das Verhalten für die identische Eingabe immer gleich. Bei zufälligen Datenstrukturen hängt das Verhalten auch von zufälligen Entscheidungen ab.

Skip List

Die Laufzeit ist im Worst-Case für Suchen/Löschen und Einfügen $\Omega(n)$. Die durchschnittliche Laufzeit für Suchen, Löschen und Einfügen ist $\Theta(log_{\frac{1}{p}} n)$. Der durchschnittliche Speicherbedarf ist $1 + pn + p^2n + ... = n \cdot \sum_{i \geq 0} p^i = \frac{n}{1-p}$ Die Idee von Skip-Listen ist, eine oder mehr "Express-Listen"hinzuzufügen. Elemente werden zu erst in den Express-Listen gesucht, ist der gesuchte Wert kleiner als das nächste Element in der Express-Liste, geht man eine Liste "tiefer". Wurde das Element gefunden, wird es ausgegeben.
Rekursiv lässt sich die Liste verbessern, in dem Man jeder Express-Liste eine neue Express-Liste mit der Hälfte der Elementen hinzufügt. Insgesamt ergibt das $\frac{n}{2} + \frac{n}{4} + ... + 2 + 1 \leq n$ zusätzliche Elemente. diese wählt man mit der Wahrscheinlichkeit p (z.B. $p = \frac{1}{2}$ aus. Die durchschnittliche Höhe h = $O(log_{\frac{1}{p}} n))$
Der Vorteil liegt darin, dass Einfügen/Löschen parallele Verarbeitung unterstützen, da die Änderungen nur sehr lokal vorgenommen werden.

Hash Tables

Mittels der Hashfunktion h aus einer Datenmenge die Daten in eine Tabelle/Array T[] hashen. Ist ein Arrayeintrag schon vorhanden, wird eine verkettete Liste erzeugt und das Elemenent parallel eingefügt. Die Laufzeit von Hash-Tables ist im Durchschnitt $\Theta(1)$ (Einfügen auch im Worst-Case). Mit verketteten Listen ist die Such- und Löschzeit im Durchschnitt $\Theta(\frac{n}{T.length})$ und die Einfügezeit im Worst-Case $\Theta(1)$. Der Speicherbedarf erhöht sich und ist bei ca $1,33n$. Es gibt Kryptographische Hash Funktionen wie MD5, SHA.1: $\{0,1\}^* \rightarrow \{0,1\}^{160}$. Aus Sicherheitsgründen wählt man eher SHA-2 oder SHA-3

Erweiterte Strukturen

Dynamische Programmierung

Wenn sich Teilprobleme überlappen, kann man das Konzept der dynamischen Programmierung anwenden.

- Struktur der optimalen Lösung charakterisieren
- rekursiv den Wert einer optimalen Lösung definieren
- Den Wert der optimalen Lösung berechnen (z.B. bottom-up Ansatz)
- zugehörige Lösung aus den berechneten Daten konstruieren

Stabzerlegungsproblem

Stangen der Länge n cm sollen so zerschnitten werden, dass der Erlös r_n maximal ist, indem die Stange in kleinere Stäbe geschnitten wird.

Länge i	Preis p_i
1	1
2	5
3	8
4	9
5	19
6	17
...	...

Eine Stange der Länge n kann in 2^{n-1} verschiedenen Weisen zerlegt werden.

Rekursiv über top-down würde das so aussehen:

```
CUT–ROD(p, n)
1   if n == 0
2       return 0
3   q=−infinity
4   for i=0 to n−1
5       q = max(q, pi+CUT–ROD(p,n−i−1))
6   return q
```

Lösung. ***via dynamischer Programmierung***
Mittels dynamischer Programmierung wollen wir CUT-ROD in einen effizienten Algorythmus verwandeln. Naiv über Rekursion wäre ineffizient, da immer die gleichen Teilprobleme gelöst werden.
Ansatz: jedes Teilproblem nur einmal lösen und öfter nachschlagen. Es wird mehr Speicherplatz genutzt um Laufzeit einzusparen (Laufzeit-Speicher-Tradeoff).

Top-Down Idee, trotzdem rekursiv aufzubauen, jedoch die Ergebnisse zu speichern und am Anfang prüfen, ob das Teilproblem bereits gelöst wurde.

Bottom-up Beginnend mit dem kleinsten Teilproblem sortieren und lösen. Werden größere Teilprobleme gelöst, wurden die kleineren Zwischenprobleme bereits gelöst. Ein Algorythmus mit Ausgabe würde dann z.B. so aussehen:

```
EXTENDED–BOTTOM–UP–CUT–ROD(p, n)
1   new arrays r[0. . n] and s[0. . n]
2   r[0] =0
3   for j = 1 to n
4       q = −∞
5       for i = 1 TO j
6           if q<p[i] + r[j−i]
7               q = p[i] + r[j−i]
8               s[j]=i
9       r[j] =q
10  return r and s
```

Greedy-Algorythmus

Entscheidet sich stets für die ihm in diesem Schritt beste Möglichkeit.

Damenproblem

Auf einem n*n Schachbrett sollen n Damen so platziert werden, dass sich gegenseitig nicht Schlagen können. Bei n=8 sind das $\binom{64}{8} = 4 \cdot 10^9$ Möglichkeiten. Da aber in jeder Spalte und Zeile nur eine Dame stehen kann, sind es nur 8! Möglichkeiten.

Ein Algorithmus für das Problem würde so aussehen:

```
PLACE–QUEENS(Q, r)
1   if r = n+1
2       return Q
3   else
4       for j = 0 to n−1
5           legal == true
6           for i = 0 to r−1
7               if (Q[i]=j) ||
                (Q[i]=j+r−i) || (Q[i]=j−r+i)
8                   legal == flase
9           if legal = true
10              Q[r] <− j
11              PLACE–QUEENS(Q, r+1)
```

Heuristiken

- Heuristik Technick um eine Suche zur Lösugn zu führen
- Metaheuristik
  - meta: über, hinter
  - Higher-level Strategie um, die Suche zu führen
  - Soll verhindern, dass man bei der Suche in einem lokalen Optimum hängen bleibt.

Bergsteigeralgorithmus

Nutzt eine Iterative Verbesserungstechnik und wendet die aktuelle Lösung im Suchraum an. In jeder Iteration wird eine neue Lösung aus der Nachbarschaft gewählt und liefert diese einen besseren Wert wird diese zur aktuellen Lösung. Sonst wird eine neue Lösung aus der Nachbarschaft gewählt und gegen die aktuelle Lösung getestet. Kann keine weitere Verbesserung erziehlt werden, terminiert der Algorithmus.

- Nachteile:
  - Terminiert bei lokalem optimum
  - Gibt keine Auskunft, wie sich die Lösung vom globalem Maximum unterscheidet
  - Optimum hängt von der initialen Konfiguration ab
- Vorteil:
  - Einfach anzuwenden

Iterative Lokale Suche

Wenn der Algorithmzs ein lokales Optimum findet, dann schaut es danach, ob es ein "nahes"Optimum gibt und dieses ggf. an. Es werden so nur Lösungen nahe der "Homebase"gewählt und der Algorithmus entscheidet, ob er die aktuelle Lösung behällt oder die neue annimt. Am besten macht man den Bergsteiger und danach einen "großen Sprung", um ein anderes Optimum zu finden, das eventuell besser ist.

- **Vortrub Funktion** soll ausreichend weit weg springen, dass es außerhalb der Nachbarschaft liegt, aber nicht soweit, dass es zufällig wird.

- **NewHomeBase Funktion** wählt die neue Startlösung aus. Es kann immer das lokale Optimum gewählt werden $NewHomeBase(H,S) = S$. Der Algorithmus nimmt die neue Lösung nur dann an, wenn die Qualität besser ist, sonst behält er die aktuelle Lösung $S, wenn Quality(S) > Quality(H) sonst H$.

Simulated Annealing

Im Gegensatz zum Bergsteiger kann entschieden werden, wie die neue Lösung die aktuelle ersetzt. Ist die neue besser, dann wird diese übernommen. Wenn die neue Lösung schlechter ist, wird diese nur mit einer Wahrscheinlichkeit von $Pr(R,S,t) = e^{\frac{Quality_{(R)} - Quality_{(S)}}{t}}$ mit $t \geq 0$ (Bruch ist negativ da R schlechter ist als S). Dadurch ist es möglich, dass der Algorithmus "Absteigt".

Tabu Search

Speichert alle bisherigen, in betracht gezogenen Lösungen in der "Tabu List"und wird diese nicht nochmal annehmen. Dadurch kann er sich jedoch von der Optimalen Lösung entfernen. Die Liste ist endlich. Wenn sie voll ist, wird der älteste Eintrag gelöscht.

Populations-basierte Methode

In den bisherigen Methaheuristiken wurde immer nur eine Lösung betrachtet, hier wird eine Stichprobe an Lösungen gleichzeitig betrachtet. Bei der Bewertung der Qualität spielt die gesamte Stichprobe jeweils eine Rolle.

Evolutionärer Algorithmus

Ein Beispiel aus der Klasse "Evolutionary Computation". Diese Klasse lässt sich in zwei Arten von Algorithmen teilen:

- generationale Algorithemn aktualisieren per Iteration die komplette Stichprobe
- steady-state Algorithmen aktualisieren nur vereinzelte Kanidaten innerhalb der Stichprobe.

Methoden baiseren auf der resampling Technik, in der neue Stichproben basierend auf den Resultaten der alten generiert werden.

```
ABSTRACT–EVOLUTIONARY–ALGORITHM
1   P <– generiere initiale Population
2   best <– [] \\ [] = Menge leer
3   REPEAT
4       AssesFitness(P)
5       for jedes individuelle P_i in P
6           if best = [] oder
            Fitness(P_i)>Fitness best
7               best <– P_i
8       P <– Join(P_1, Breed(P))
9   UNTIL best solution or timeout
10  return best
```

Breed Funktion benutzt die Fitnessinformation um die neue Stichprobe zu erstellen. Die Join Funktion fügt diese neue Population der Menge hinzu.

Evolutionäres. Initialisiert wird die Stichprobe durch zufälliges wählen der Elemente. Sieht man in der Rohmenge "gute"Regionen, kann man die zufällige Generierung der Stichprobe beeinflussen, um in dieser Region zu landen. Diversität der Population durch Sicherstellung, dass alle Elemente in der Population einzigartig sind.
Wenn ein neues Individuum zufällig gewählt wird: Vergleichen aller biserigen Individuen in Popolation ist ein Algorithmus, der alle Möglichkeiten miteinander vergleicht ($\Theta(n^2)$). Besser ist eine Hashtable zu nutzen und zu checken, ob das Individuum schon in der Hashtable ist ($O(n)$).

Evolutionsstrategien

Idee: Generiere eine Population zufällig. Beurteile die Qualität jedes Individuums. Lösche alle bis auf die μ besten Individuen. Dann generiert man λ/μ-viele Nachfahren pro bestem Individuum und ersetzt die Individuen durch die Nachfahren per Join-Funktion.

Amortisierte Analyse

Algorithmen führen eine Folge von Operationen auf Datenstrukturen aus. Für die Laufzeit-Analyse haben wir die Kosten abgeschätzt, die eine Folge von n Operationen im worst-case benötigt. Dies ist die Obere Schranke der Gesamtkosten. durch die Amortisierte Analyse lassen sich worstcase Kosten viel genauer abschätzen. Voraussetzugn hierfür ist, dass nicht alle Operationen in der Operationenfolge gleich teuer sind, sondern dass die Kosten von dem Zustand der Datenstruktur abhängen. die Amortisierte Analyse garantiert die mittlere Performanz jeder operation im schlechtesten Fall.

Graphen

Graph G=(V,E)
V beschreibt die Knotenmenge
E beschreibt die Kantenmenge
(u,v) ∈ E beschreibt die Kante von Knoten u zu v
Pfade: Knoten v ist von Knoten u im Graphen G=(V,E) erreichbar,wenn es Pfad $(w_1,...,w_k) \in V^k$ gibt, so dass $(w_j,w_j \in)$ E für j=1,2,...k-1 und w_1 = u und w_k=v.
Man kann Graphen auch als Bäume darstellen, wenn es einen Knoten gibt der als Wurzel agieren kann, sodass jeder Knoten per eindeutigen Pfad erreichbar ist.
Subgraphen müssen von der gleichen Art ihrer Graphen sein, sie sind also Teilgraphen die Kanten und Knoten auslassen können.

(Endliche) Gerichtete Graphen

Ein (endlicher) gerichteter Graph G=(V,E) besteht aus
(1) einer (endlichen) Knotenmenge V(„vertices“)
(2) einer (endlichen) Kantenmenge EVV(„edges“)
(3) Zyklen sind möglich z.B 1,5,6,2,1
(4) Schleifen sind möglich sowie isolierte Knoten(Knoten ohne Kanten)
(5) Es gibt keine Mehrfachkanten zwischen Knoten (6) ist stark zusammenhängend wenn jeder Knoten von jedem anderen Knoten (gemäß Kantenrichtung) aus erreichbar ist

Ungerichteter Graph

Ein (endlicher) ungerichteter Graph G=(V,E)
(1) einer (endlichen) Knotenmenge V(„vertices“)
(2) einer (endlichen) Kantenmenge EVV(„edges“) **so dass (u,v)E dasselbe ist wie (v,u)**, alternativ u,v
(3) ist zusammenhängend wenn jeder Knoten von jedem anderem Knoten aus erreichbar ist.

Darstellung

als Adjazenzmatrix. Mann kann die Verbindung von Graphen in einer Matrix (V x V) darstellen. Wenn Knoten verbunden sind z.B. Knoten 1 und 5 also (1,5) kann als Zeile 1 Spalte 5 interpretiert werden. Bei ungerichteten Graphen ist die Matrix spiegelsymmetrisch zur Hauptdiagonalen.

als Adjazenzliste. Bei der Darstellung als Liste kann man ein Array von verketteten Listen verwenden. Dabei kann man das Array sortiert oder unsortiert erstellen. Da es eine verkette Liste ist zeigt der Knoten 4 z.B auf Knoten 5 dann.

Gewichtete Graphen

Die Kanten des Graphen haben hier ein Gewicht bzw. Entfernung die z.B. bei der Berechnung des kürzesten Pfades miteinberchenet werden müssen/können.

Search-Methoden

BFS Breadth-First-Search. Die Idee dahinter ist dass man zuerst die umliegenden Nachbarn eines Knoten absucht und dann die Nachbarn der Nachbarn usw.Laufzeit =O(|V|+|E|)

```
BFS(G,s) //G=(V,E), s=source node in V
//WHITE =Knoten noch nicht besucht
//GRAY =in Queue fuer naechsten Schritt
//BLACK =fertig
1    FOREACH u in V-{s} DO
2         u.color=WHITE;
          u.dist=INT.MAX; u.pred=NIL;
3    s.color=GRAY; s.dist=0; s.pred=NIL;
4    newQueue(Q);
5    enqueue(Q,s);
6    WHILE !isEmpty(Q) DO
7         u=dequeue(Q);
8         FOREACH v in adj(G,u) DO
//adj(G,u) gibt alle Nachbarknoten von u
9              IF v.color==WHITE THEN
10                  v.color=GRAY;
                    v.dist=u.dist+1;
                    v.pred=u;
11                  enqueue(Q,v);
12        u.color=BLACK;
```

Korrektheit
Sei G=(V,E) ein Graph mit Knoten s ∈ V. Dann gilt nach Terminierung von BFS(G,s) für jeden Knoten von s aus erreichbaren Knoten v, dass shortest(s,v)=v.dist. Für v ≠ s ist ein kürzester Pfad durch einen kürzesten Pfad von s nach vpred und der Kante (v.pred,v) gegeben.
Im ersten Schritt werden genau die Knoten besucht, die von s aus über eine Kante erreicht werden können; diese Knoten erhalten dist=1.
Im zweiten Schritt werden nur die Knoten besucht die in zwei oder mehr Schritten von s aus erreichbar sind; diese erhalten dist=2 usw.
Für u=v.predist, da vvon uaus per Kante (u,v)∈E besucht wurde, der Pfad von s nach u und dann zu v ein Pfad der Länge:
1+schortest(s,u)=1+u.dist=v.dist=shortest(s,v)

Kürzesten Pfad ausgeben

```
PRINT-PATH(G,s,v)
//assumes that BFS(G,s) has already
//been executed
1    IF v==s THEN
2         PRINT s
3    ELSE
4         IF v.pred==NIL THEN
5              PRINT "no␣path␣from␣s␣to␣v"
6         ELSE
7              PRINT-PATH(G,s,v.pred);
8              PRINT v;
```

DFS Depth-First-Search. Besuche zuerst alle noch nicht besuchten Nachfolgeknoten („Laufe so weit wie möglich weg von aktuellem Knoten“). Laufzeit =O(|V|+|E|). DFS-Wald = Menge von DFS-Bäumen, wobei der Baum nicht unbedingt den kürzesten Pfad angibt.

```
DFS(G) //G=(V,E)
1    FOREACH u in V DO
2         u.color=WHITE;
3         u.pred=NIL;
4         time=0;//globale Variable
5    FOREACH u in V DO
6         IF u.color==WHITE THEN
7              DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u)
1    time=time+1;
2    u.disc=time; //discovery time
3    u.color=GRAY;
4    FOREACH v in adj(G,u) DO
5        IF v.color==WHITE THEN
6            v.pred=u;
7            DFS-VISIT(G,v);
8    u.color=BLACK;
9    time=time+1;
10   u.finish=time; //finish time
```

Kantenarten der DFS Bäume
Sei G_{pred} der Subgraphbaum von unserem gerichtete, Graphen G:
Baumkanten: alle Kanten in in G_{pred}
Vorwärtskanten: alle Kanten in G zu Nachkommen in G_{pred} die nicht Baumkante
Rückwärtskanten: alle Kanten in G zu Vorfahren in G_{pred} die nicht Baumkante (inkl. Schleifen)
Kreuzkanten: alle anderen Kanten in G
Sei (u,v)gerade betrachte Kante im DFS-Algorithmus. Dann ist (u,v)...
eine Baumkante, wenn v.color==WHITE
eine Rückwärtskante, wenn v.color==GRAY
eine Vorwärtskante, wenn v.color==BLACK und u.disc<v.disc
eine Kreuzkante, wenn v.color==BLACK und v.disc<u.disc

In einem ungerichteten Graphen gibt es nur Baum und Rückwärtskanten:
Vorwärts-und Kreuzkanten haben v.color==BLACK
Sei u gerade aktiv (grau). Betrachtete neü Kante u,vkann nur Vorwärts-oder Kreuzkante werden, wenn v schon abgeschlossen(schwarz).
1.Fall: u.disc<v.disc:
Kann nur passieren, wenn v von u aus durch anderen Pfad bereits erreicht und v in dem Moment grau wurde. Dann wäre aber Kante v,u bereits bei v Rückwärtskante geworden.
2.Fall: u.disc>v.disc
Dann hätte u bereits besucht werdenmüssen, als v aktiv (grau) war:
Direkt (u und v sind Nachbarn):u noch weiß und v,u bereits Baumkante.
Indirekt (u und v haben Zwischenknoten) u wird grau und u,v bereits Rückwärtskante.

Anwendung von DFS

Topologisches Sortieren. Topologisches Sortieren ist nur für gerichtete Graphen ohne Zyklen(auch "dag"genannt). Topologische Sortierung eines dag G=(V,E): Knoten in linearer Ordnung, so dass für alle Knoten u,v∈V gilt, dass u vor v in der Ordnung kommt, wenn (u,v)∈ V.

```
TOPOLOGICAL-SORT(G) // G=(V,E) dag
1    newLinkedList(L);
2    runDFS(G) but, eachtime a nodeisfinished,inser
3    returnL.head
```

Laufzeit= $O(|V|+|E|)$
Korrektheit
Es genügt zu zeigen, dass jede von DFS inspizierte Kante (u,v) ∈V erfüllt: v.finish<u.finish, so dass u in der Liste dann vor v eingefügt wird.
1.Fall: v bereits grau -> Würde Rückwärtskante erzeugen,d.h. der Graph hätte einen Zyklus (Widerspruch).
2.Fall: v noch weiß -> Erzeugt Baumkante, also wird v Nachfahre von u und v.finish<u.finish, da Aufrufs-Stack nicht zu u zurückkehrt, bevor v abgeschlossen.
3.Fall: v schwarz -> Dann v.finish bereits gesetzt, während u.finish erst später gesetzt wird, also v.finish<u.finish.

Starke Zusammenhangskomponenten. Eine starke Zusammenhangskomponente (SCC) eines gerichteten Graphen G=(V,E) ist eine Knotenmenge C ⊆ V, so dass

- a) es zwischen je zwei Knoten u,v ∈ C einen Pfad von u nach v gibt, und
- b) es keine Menge D ⊆ V mit C echte Teilmenge von V gibt, für die (a) auch gilt

(C ist maximal.)
Eigenschaften Verschiedene SCCs C,D sind disjunkt,sonst gäbe es v∈C∩D und für beliebige u∈C und w∈D auch einen Pfad von u nach w über v (und umgekehrt),somit wären C und D identisch.
Zudem sind zwei SCCs nur in eine Richtung verbunden. Also nur C nach D zum Beispiel und nicht D nach C.

Algorithmus Der Ansatz hier ist das man DFS einmal auf den Graphen G und dann nochmal auf den transponierten Graphen von G, also G^T (Hierbei werden nur die Richtungen der Kanten vertauscht) laufen lässt.

```
SCC(G) // G=(V,E) directedgraph
1    runDFS(G)
2    computeGT
3    runDFS(GT) but visit vertices in main
       loop in descending finish time from 1
4    output each DFS tree in 3 as one SCC
//Zusatz fuer Zeile 3
DFS(G) //G=(V,E)
1    FOREACH u in V DO
2        u.color=WHITE;
3        u.pred=NIL;
4    time=0;
5    FOREACH u in V DO
6        IF u.color==WHITE THEN
```

```
7                DFS-VISIT(G,u)
```

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, wenn T azyklisch ist und alle Knoten verbindet. Der Spannbaum ist minimal, wenn w(T)=$\sum w(u,v)$ wobei $u,v \in E_T$.

Generische Methode. A ist die Teilmenge der Kanten eines MST
Kante (u,v) ist sicher („safe“) für A, wenn A∪(u,v) noch Teilmenge eines MST ist
genericMST(G,w) // G=(V,E) undirected, connected graph w weight function
1 A= ∅
2 WHILE A does not form a spanning tree for G DO
3 find safe edgeu,v for A
4 A = A∪u,v
5 return A

Korrektheit.

- **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 ∪ (u,v) noch Teilmenge eines MST ist), ist am Ende der WHILE-Schleife A ein MST.

Terminologie. Sei A Teilmenge eines MST,(S,V-S) Schnitt, der A respektiert, und u,v eine leichte Kante, die den Schnitt überbrückt.Dann ist u,v sicher fuer A.

- Schnitt (S,V-S)partitioniert Knoten des Graphen in zwei Mengen.
- u,vüberbrückt Schnitt (S,V-S),wenn u∈S und v∈V-S.
- Schnitt (S,V-S)respektiert A ⊆ E ,wenn keine Kante u,vaus A den Schnitt überbrückt
- u,vleichte Kante für (S,V-S),wenn w((u,v))minimal für alle den Schnitt überbrückenden Kanten

Algorithmus von Kruskal

Dieser Algorithmus laesst parallel mehrere Unterbaeume eines MST wachsen.
Laufzeit ist $O(|E|*\log|E|)$ oder auch $O(|E|*\log|V|)$, da $|V|-1 \leq |E| \leq |V|^2$ und somit $\log|E| = \Theta(\log|V|)$

```
MST-Kruskal(G,w) // G=(V,E) undirected,
          connected graph w weight funct
1    A=∅
2    FOREACH v in V DO set(v)={v};
3    Sort edges according to weight in
          nondecreasing order
4    FOREACH
          {u,v} in E according to order DO
5         IF set(u)!=set(v) THEN
     //Wenn gleich, dann schon verbunden.
6              A = A∪{{u,v}}
7              UNION(G,u,v);
8    return A
```

Notizen:
UNION(G,u,v) setzt set(w)=set(u)∪ set(v) für alle Knoten w∈ set(u)∪ set(v)

Korrektheit. Jede ausgewählte Kante u,v mit set(u) != set(v) ist leicht für Schnitt (set(u),V-set(u)).Schnitt respektiert A und somit ist die Kante auch sicher fuer A.

Algorithmus von Prim

Dieser Algorithmus konstruiert einen MST Knoten für Knoten. Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zuzusammenhängender Menge hinzu. Auswahl der naechsten Kante gemaeß key-Wert, der stets aktualisiert wird.
A implizit definiert durch A=v,v.pred|v∈ V-(rcupQ)
Laufzeit ist $O(|E|+|V|*\log|V|)$ (mit vielen Optimierungen,speziell Fibonacci-Heaps)

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

Korrektheit. Kanten in A laufen nur zwischen den bereits aufgesammelten Knoten in V-Q. Folglich respektiert der Schnitt (Q,V-Q)die Menge A. Alle Knoten v∈Q enthalten als Wert v.key immer das kleinste Kantengewicht zu einem bereits aufgesammelten Knoten v.pred in V-Q. Daher beschreibt der in Schritt 4 ausgewaehlte Knoten u eine überbrückende, leichte Kante(u,u.pred).

Kürzester Weg,gerichteter Graph

SSSP. Single-Source ShortestPath (SSSP) findet von Quelle s aus jeweils den gemaeß Kantengewicht kürzesten Pfad zu allen anderen Knoten. Die Unterschiede zu DFS und BFS sind dass Sie beide keine Kantengewichte unterstützen, da der BFS z.B. nur den kürzesten Kantenweg findet, aber nicht den kürzesten im Hinblick auf Gewichte. Der Unterschied zu MST ist dass dieser das Gewicht im ganzen Baum versucht zu minimieren. Problem bei negativen Kantengewichten gibt es bei Zyklen im Graph einen immer kleineren Pfad, da man die Zyklen mehrmals durchlaufen kann.

Lockerung bzw. Relax. Idee: verringere aktuelle Distanz von Knoten u, wenn durch Kante (u,v)kürzere Distanz erreichbar:
relax(G,u,v,w) 1 IF v.dist> u.dist+ w((u,v)) THEN
2 v.dist=u.dist+ w((u,v));
3 v.pred=u;

Bellman Ford Algorithmus. Es funktioniert allgemein, anders als die unteren Algorithmen. Es betrachtet die Wirkung auf kürzesten Pfades von s nach z, bei der ersten Iteration der FOR-Schleife wird der erste Schritt zum nächsten Knoten aufgefasst. Bei der zweiten Iteration erfasst dann den zweiten Schritt von s zum zweiten Knoten usw. Laufzeit ist *Theta*(|V|*|E|)

```
Bellman-Ford-SSSP(G,s,w)
1   initSSSP(G,s,w);
2   FOR i=1 TO |V|-1 DO
3       FOREACH (u,v) in E DO
4           relax(G,u,v,w);
5   FOREACH (u,v) in E DO//prueft ob ein negativer
                          //Zyklus erreichbar ist
6       IF v.dist> u.dist+w((u,v)) THEN
7       return false;
8   return true;

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

Korrektheit: Wir betrachten Prüfungsschritte(Zeilen) 5-7

- Fall 1: Wenn keine „negativen Zyklen“ erreichbar, dann auch keine Rückgabe false
- Fall 2: Wenn „negativen Zyklen“ erreichbar,dann Rückgabe false

TOPO-Sort. Topologisches Sortieren fuer DAG(Directed-Acyclic-Graph) mit Gewichten. Laufzeit ist *Theta*(|V|+|E|)

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

Korrektheit siehe Bellman-Ford-Algorithmus

Dijkstra-Algorithmus. Voraussetzung:w((u,v))≥ 0 für alle Kanten. Laufzeit ist *Theta*(|V|*log|V|+|E|) mittels Fibonacci-Heaps.

```
Dijkstra-SSSP(G,s,w)
1   initSSSP(G,s,w);
2   Q=V; //letS=V-Q
3   WHILE !isEmpty(Q) DO
4       u=EXTRACT-MIN(Q); //wrt. dist
5       FOREACH v in adj(u) DO
6   relax(G,u,v,w);
```

Korrektheit Für jeden betrachteten Knoten u in der WHILE-Schleife gilt: u.dist=shortest(u,v). Angenommen, u wäre erster Knoten, bei dem nicht der Fall. Ins besondere u≠ s und S ≠ ∅ da s.dist=0 korrekt ist. Betrachte kürzesten Pfad s->u mit „erstem“ Knoten y nicht in S,und Vorgängerknoten x in S. Es gilt x.dist=shortest(s,x),da u der erste Knoten ist, bei dem diese Gleichung nicht gilt. Da Kante (x,y)in dem Moment, als x zu S hinzugefügt wurde, auch „relaxed“ wurde,gilt auch y.dist=shortest(s,y). Da nur positive Kantengewichte gilt shortest(s,y)≤shortest(s,u) und damit y.dist =shortest(s,y)≤shortest(s,u) ≤u.dist. Andererseits wurde u vor y für S ausgewählt, also u.dist≤ y.dist und y.dist= shortest(s,y)≤shortest(s,u) ≤ u.dist. Folglich y.dist = shortest(s,y) =shortest(s,u)= u.dist, da u.dist ≤ y.dist und y.dist=shortest(s,y)≤shortest(s,u)≤u.dist

Maximaler Fluss in Graphen

Kanten in Graphen haben (aktullen) Flusswert und (maximale) Kapazität. Jeder Knoten (außer s und t) hat den gleichen eingehenden und ausgehenden Fluss. Das Ziel ist es, den maximalen Fluss zu von s (source) zu t (sink) zu finden.

Netzwerkflüsse. Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph G=(V, E) mit Kapazität c, mit c(u, v) ≥ 0 für (u, v) ∈ E und c(u, v) = 0 für (u,v) not ∈ E, mit zwei Knoten s,t ∈ V, sodass jeder Knoten aus s erreichbar ist und jeder Knoten t erreichen kann.
Ein Fluss $f : VxV \to \mathbb{R}$ für ein Flussnetzwerk G = (V,E) mit Kapazität c, s, t erfüllt $0 \leq f(u,v) \leq c(u,v)$ für alle $u, v \in V$ sowie für alle $u \in V - \{s,t\}$:

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$$

Der Wert $|f|$ eines Flusses $f : VxV \to \mathbb{R}$ für ein Fluss-

netzwerk G=(V,E) mit Kapazität c, source s, sink t ist

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, s)$$

Transformationen. Anitparallle Kanten eliminieren durch erstellen eines neuen Knotens:
A <-(2,5)–|–(1,6)-> B wird zu A –(1,6)->C–(1,6)->B–(2,5)->A
Quellen werden vereinigt, indem man eine neue Quelle erstellt, die zu allen vorherigen Quellen führt und einen neuen sink erstellt, den alle vorherigen sinks erreichen.

Ford-Fulkerson-Methode. Idee ist es von s nach t einen noch erweiterbaren Pfad zu suchen. Diesen Pfad suchen wir im "Restkapazitäts"-Graph G_f, der die möglichen Zu- und Abflüsse beschreibt.

Reste. Die Restkapazität besteht
$c_f(u, v) = c(u, v - f(u, v)$ falls $(u, v) \in E$
$f(v, u)$ falls $(v, u) \in E$
0 sonst

- 1. bedeutet, wieviel eigehenden Fluss über (u,v) könnte man noch zu v hinzufügen?
- 2. bedeutet, wieviel abgehenden Fluss über (v,u) könnte man wegnehmen und damit quasi zu v hinzufügen?

$G_f = (V, E_f) mit E_f = \{(u, v) \in VxV | c_f(u, v) > 0\}$. Im Restkapazitätsgraph sind antiparallele Kanten erlaubt. Um den Graphen zu nutzen: 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.

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

NP

nur einfache Probleme sind in polynomialer Zeit loesbar! Es gibt Probleme, die nur in superpolynomieller Zeit gelöst werden können. Diese Probleme nennt man "harte"Probleme. So ein Problem ist "effizient Lösbar"gdw. es in polynomieller Zeit lösbar ist.

Logarithmus-Gesetze

[3]

- **Produktregel** $\log_a(u) + \log_a(v) = \log_a(u \cdot v)$
- **Quotientenregel** $\log_a(u) - \log_a(v) = \log_a(\frac{u}{v})$
- **Potenzregel**
  - $\log_a(x^n) = n \cdot \log_a(x)$
  - $\log_a(\sqrt[n]{x}) = \log_a(x^{\frac{1}{n}}) = \frac{1}{n} \cdot \log_a(x)$
- **Umrechnung der Logarithmusbasis** $\log_n(x) = \frac{\log_m(x)}{\log_n(n)}$
- **Weitere Eigenschaften von Logarithmen**
  - $\log_n(n) = 1$
  - $\log_n(1) = 0$
  - $\log_n(n^x) = x$
  - $n^{\log_n(x)} = x$
- **Küren von Exponenten** $x_1^{x_2} = b^{x_2 \cdot \log_b(x_1)}$
  Dieses Gesetz kann verwendet werden, um beliebige Exponenten auf eine bestimmte Basis umzuschreiben. Will man beispielsweise eine Funktion ableiten, die sowohl x in der Basis als auch im Exponenten hat, so kann diese Regel angewendet werden, um die Funktion zur Basis e umzuschreiben. Die entstandene Exponentialfunktion lässt sich dann mit einfachen Mitteln ableiten. Wählt man als Basis e, so erhält man: $x_1^{x_2} = e^{x_2 \cdot \ln(x_1)}$
- **Antilogarithmus**
  - **Multiplikation** $x \cdot y = a^{\log_a x} \cdot a^{\log_a y} = a^{\log_a x + \log_a y}$
  - **Division** $\frac{x}{y} = \frac{a^{\log_a x}}{a^{\log_a y}} = a^{\log_a x - \log_a y}$

[3] https://www.geeksforgeeks.org/avl-tree-set-2-deletion/