

Algorithmen Zusammenfassung

Stable-Matching-Problem

Gale-Shapley-Algorithmus

Weitere Algorithmen zum Stable-Matching-Problem findet man bei [Analyse von Algorithmen](#)
[> Stable-Matching-Problem](#)

Gale–Shapley-Algorithmus (GS-Algorithmus)

Gale-Shapley-Algorithmus:

```
Kennzeichne jede Familie/jedes Kind als frei
while ein Kind ist frei und kann noch eine Familie wählen
    Wähle solch ein Kind s aus
    f ist erste Familie in der Präferenzliste von s,
    die von s noch nicht gewählt wurde
    if f ist frei
        Kennzeichne s und f als einander zugeordnet
    elseif f bevorzugt s gegenüber ihrem aktuellen Partner s'
        Kennzeichne s und f als einander zugeordnet und s' als frei
    else
        f weist s zurück
```

Analyse von Algorithmen

Binärsuche

Logarithmisches Wachstum: $O(\log n)$

Logarithmisches Wachstum: Wenn z.B. die Eingabegröße in jedem Schritt halbiert wird.

Binäre Suche: Binäre Suche nach einem gegebenen Wert in einem aufsteigend sortierten Array A (siehe VU Einführung in die Programmierung 1)

```
Ermittle den mittleren Index s im Array A
if A[s] = gesuchter Wert
    Gib aus, dass Wert gefunden wurde
elseif gesuchter Wert ist kleiner als A[s]
    Wende binäre Suche auf das Teilarray links von s an
elseif gesuchter Wert ist größer als A[s]
    Wende binäre Suche auf das Teilarray rechts von s an
```

Maximum Suche

Lineares Wachstum: $O(n)$

Lineares Wachstum: Laufzeit ist proportional zur Eingabegröße.

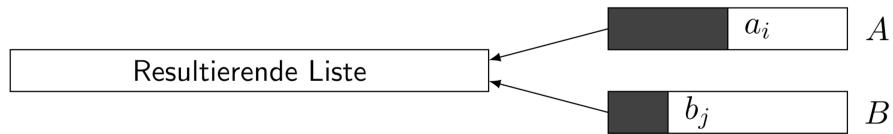
Maximumsuche: Ermittle das Maximum von n Zahlen a_1, \dots, a_n .

```
max ← a1
for i ← 2 bis n
    if ai > max
        max ← ai
```

Verschmelzen zweier Listen

Lineares Wachstum: $O(n)$

Merge: Verschmelze zwei sortierte Listen $A = a_1, a_2, \dots, a_n$ und $B = b_1, b_2, \dots, b_n$ zu einer sortierten Liste.



```
i ← 1, j ← 1
while beide Listen noch nicht komplett bearbeitet
    if ai ≤ bj
        Füge ai zur Ergebnisliste hinzu und erhöhe i
    else
        Füge bj zur Ergebnisliste hinzu und erhöhe j
    Füge den Rest der noch nicht komplett bearbeiteten Liste
    zur Ergebnisliste hinzu
```

Alle Elemente in einem Array mit allen anderen Elementen vergleichen

Die optimierte Version des dichteste Punktpaar Problem findet man bei [Divide-and-Conquer > Dichteste Punktpaar](#)

Quadratisches Wachstum: $O(n^2)$

Quadratisches Wachstum: Betrachte alle Paare von Elementen.

Dichtestes Punktpaar: Gegeben sei eine Liste von n Punkten in einer Ebene $(x_1, y_1), \dots, (x_n, y_n)$ und es sollen die zwei am dichtesten beieinander liegenden Punkte gefunden werden.

$O(n^2)$ Lösung: Überprüfe alle Paare von Punkten.

```
min ←  $(x_1 - x_2)^2 + (y_1 - y_2)^2$ 
for  $i \leftarrow 1$  bis  $n - 1$ 
    for  $j \leftarrow i + 1$  bis  $n$ 
         $d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$ 
        if  $d < min$ 
             $min \leftarrow d$ 
```

■ Es muss nicht die Wurzel gezogen werden

Bestimmen, ob alle Teilmengen disjunkt sind

Kubisches Wachstum: $O(n^3)$

Kubisches Wachstum: Zähle alle Dreiergruppen von Elementen.

Disjunkte Mengen: Gegeben seien n Mengen S_1, \dots, S_n , wobei jede Menge eine Teilmenge von $\{1, 2, \dots, n\}$ ist. Existiert ein Paar von Mengen, das disjunkt ist?

$O(n^3)$ Lösung: Bestimme für jedes Paar von Mengen, ob es disjunkt ist.

```
for i ← 1 bis n - 1
    for j ← i + 1 bis n
        foreach Element p von Si
            Bestimme ob p auch in Sj vorhanden ist
        if kein Element von Si ist in Sj vorhanden
            Gib aus, dass Si und Sj disjunkt sind
```

Teilsummenproblem mit k Zahlen

Polynomielle Laufzeit: $O(n^k)$ Wachstum

Teilsummenproblem mit k Zahlen: Gegeben seien n verschiedene ganze Zahlen in einem Array A . Gibt es genau k Zahlen ($k < n$), sodass die Summe dieser Zahlen einer gegebenen Zahl x entspricht?

Beispiel: $O(n^4)$ Lösung für $k = 4$ mit Ausgabe

```
for i ← 0 bis n - 4
    for j ← i + 1 bis n - 3
        for k ← j + 1 bis n - 2
            for l ← k + 1 bis n - 1
                if (A[i] + A[j] + A[k] + A[l]) = x
                    Gib A[i], A[j], A[k] und A[l] aus
```

Teilsummenproblem

Exponentielles Wachstum: $O(c^n)$

Teilsummenproblem (Subset Sum): Gegeben sei eine Menge von ganzen Zahlen. Gibt es eine Untermenge dieser Zahlen, deren Elementsumme einer vorgegebenen Zahl x entspricht?

Lösung:

```
foreach Teilmenge  $S$  von Zahlen  
    Überprüfe, ob die Summe der Elemente in  $S$  gleich  $x$  ist
```

Exponentielles Wachstum: Eine endliche Menge mit n Elementen hat genau 2^n Teilmengen und jede dieser Teilmengen muss untersucht werden.

Stable-Matching-Problem

Array Ranking

Erstellung des Arrays Ranking

Für jede Familie (Zeile): Erzeuge eine **inverse Liste** der Präferenzliste der Familien.

Beispiel: Abel (hat Nummer 0)

Präferenz	0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5	1
Ranking	0	1	2	3	4	5	6	7
0	3	7	1	4	5	6	2	0

Abel bevorzugt Kind 2 gegenüber 5 da $\underbrace{\text{Ranking}[0, 2]}_1 < \underbrace{\text{Ranking}[0, 5]}_6$

```
for i ← 0 bis n - 1
    for j ← 0 bis n - 1
        Ranking[i, FPref[i, j]] ← j
```

Allgemeiner Algorithmus

Pseudocode mit Arrays

Gegeben: Arrays SPref und FPref

```
for i ← 0 bis n – 1
    for j ← 0 bis n – 1
        Ranking[i,FPref[i,j]] ← j
for i ← 0 bis n – 1
    Next[i] ← 0
    Current[i] ← -1
SFree ← Liste aller Kinder
while SFree ist nicht leer
    s ← erstes Element aus SFree, lösche erstes Element aus SFree
    f ← SPref[s,Next[s]]
    s' ← Current[f]
    if s' = -1
        Current[f] ← s
    elseif Ranking[f,s] < Ranking[f,s']
        Current[f] ← s
        Füge s' in SFree an erster Stelle ein
    else
        Füge s in SFree an erster Stelle ein
    Next[s] ← Next[s] + 1
```

Selectionsort

Selectionsort (Sortieren durch Minimumssuche)

Selectionsort: Selectionsort sucht in jeder Iteration das kleinste Element in der noch unsortierten Teilfolge, entnimmt es und fügt es dann an die sortierte Teilfolge an.

Beispiel: Implementierung mit einem Array A mit n Elementen.

```
Selectionsort(A):
    for i ← 0 bis n - 2
        smallest ← i
        for j ← i + 1 bis n - 1
            if A[j] < A[smallest]
                smallest ← j
    Vertausche A[i] mit A[smallest]
```

Insertionsort

Insertionsort (Sortieren durch Einfügen)

Insertionsort: Insertionsort entnimmt der unsortierten Teilfolge ein Element und fügt es an richtiger Stelle in die (anfangs leere) sortierte Teilfolge ein.

Beispiel: Implementierung mit einem Array A mit n Elementen.

```
Insertionsort(A):
    for i ← 1 bis n - 1
        key ← A[i], j ← i - 1
        while j ≥ 0 und A[j] > key
            A[j + 1] ← A[j]
            j ← j - 1
        A[j + 1] ← key
```

Graphen

Ungerichtete Graphen: Kanten ausgeben

Ungerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```
for u ← 0 bis n - 2
    for v ← u + 1 bis n - 1
        if M[u, v] = 1
            Gib Kante (u, v) aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for u ← 0 bis n - 1
    foreach Kante (u, v) inzident zu u
        if u < v
            Gib Kante (u, v) aus
```

Gerichtete Graphen: Kanten ausgeben

Gerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```
for  $u \leftarrow 0$  bis  $n - 1$ 
  for  $v \leftarrow 0$  bis  $n - 1$ 
    if  $M[u, v] = 1$ 
      Gib Kante  $(u, v)$  aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for  $u \leftarrow 0$  bis  $n - 1$ 
  foreach Kante  $(u, v)$  inzident zu  $u$ 
    Gib Kante  $(u, v)$  aus
```

Breitensuche: Implementierung mit einer Queue

Breitensuche: Implementierung mit einer Queue

Implementierung: Array Discovered, Queue Q , Graph $G = (V, E)$, Startknoten s .

```
BFS( $G, s$ ):  
    Discovered[ $s$ ]  $\leftarrow$  true  
    Discovered[ $v$ ]  $\leftarrow$  false für alle anderen Knoten  $v \in V$   
     $Q \leftarrow \{s\}$   
    while  $Q$  ist nicht leer  
        Entferne ersten Knoten  $u$  aus  $Q$   
        Führe Operation auf  $u$  aus (z.B. Ausgabe)  
        foreach Kante  $(u, v)$  inzident zu  $u$   
            if !Discovered[ $v$ ]  
                Discovered[ $v$ ]  $\leftarrow$  true  
                Füge  $v$  zu  $Q$  hinzu
```

Breitensuche: Ermitteln der Ebenen

Breitensuche: Ermitteln der Ebenen

Anwendung von BFS: Ermitteln der Ebene jedes einzelnen Knotens.

Implementierung: Array Level, Queue Q , Graph $G = (V, E)$, Startknoten s .

```
BFS( $G, s$ ):  
    Level[ $s$ ]  $\leftarrow 0$   
    Level[ $v$ ]  $\leftarrow -1$  für alle anderen Knoten  $v \in V$   
     $Q \leftarrow s$   
    while  $Q$  ist nicht leer  
        Entferne ersten Knoten  $u$  aus  $Q$   
        foreach Kante  $(u, v)$  inzident zu  $u$   
            if Level[ $v$ ] == -1  
                Level[ $v$ ]  $\leftarrow$  Level[ $u$ ] + 1  
                Füge  $v$  zu  $Q$  hinzu
```

Tiefensuche (Depth First Search, DFS)

Tiefensuche (*Depth First Search, DFS*)

DFS Ansatz: Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.

DFS Algorithmus: Startknoten s , globales Array Discovered, Graph $G = (V, E)$.

```
DFS( $G, s$ ):  
    Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
    DFS1( $G, s$ )  
  
DFS1( $G, u$ ):  
    Discovered[ $u$ ]  $\leftarrow$  true  
    Führe Operation auf  $u$  aus (z.B. Ausgabe)  
    foreach Kante  $(u, v)$  inzident zu  $u$   
        if !Discovered[ $v$ ]  
            DFS1( $G, v$ )
```

Zusammenhangskomponenten zählen

Zusammenhangskomponenten zählen

DFSNUM Algorithmus: Startknoten s , globales Array Discovered, Graph $G = (V, E)$.

```
DFSNUM( $G$ ):  
    Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
     $i \leftarrow 0$   
    foreach Knoten  $v \in V$   
        if Discovered[ $v$ ] = false  
             $i \leftarrow i + 1$   
            DFS1( $G, v$ )  
return  $i$ 
```

Gerichteter azyklischer Graph – Erkennen eines DAG mittels wiederholtem Löschen von Kanten

Gerichteter azyklischer Graph – Erkennen eines DAG mittels wiederholtem Löschen von Kanten

```
while G hat mindestens einen Knoten
    if G hat eine Quelle
        Wähle eine Quelle v aus
        Gib v aus
        Lösche v und alle inzidenten Kanten aus G
    else return G ist kein DAG
return G ist ein DAG
```

Hinweis:

- Ein Knoten kann im Lauf des Algorithmus zur Quelle werden.
- Falls G ein DAG ist, gibt dieser Algorithmus eine topologische Sortierung aus.

Topologische Sortierung

Topologische Sortierung

Algorithmus: Effiziente Implementierung des Löschalgorithmus: Löschen von Knoten wird mittels Hilfsarray count simuliert. Es wird zusätzlich eine anfangs leere Liste L verwendet.

```
foreach  $v \in V$ 
    count[ $v$ ]  $\leftarrow 0$ 
foreach  $v \in V$ 
    foreach Kante  $(v, w) \in E$ 
        count[ $w$ ]  $\leftarrow$  count[ $w$ ]+1
foreach  $v \in V$ 
    if count[ $v$ ] = 0
        Gib  $v$  zur Liste  $L$  am Anfang hinzufügen
while  $L$  ist nicht leer
    Sei  $v$  erstes Element in  $L$ , lösche  $v$  aus  $L$ 
    Gib  $v$  aus
    foreach Kante  $(v, w) \in E$ 
        count[ $w$ ]  $\leftarrow$  count[ $w$ ]-1
        if count[ $w$ ] = 0
            Gib  $w$  zur Liste  $L$  am Anfang hinzufügen
```

Dijkstra-Algorithmus

Die effizientere Version findet man bei [Graphen > Dijkstra-Algorithmus Effizientere Variante](#)

Dijkstra-Algorithmus

Algorithmus: Arrays Discovered und d , Graph $G = (V, E)$, Liste L , Startknoten s .

```
Dijkstra( $G, s$ ):  
    Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
     $d[s] \leftarrow 0$   
     $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$   
     $L \leftarrow V$   
    while  $L$  ist nicht leer  
        wähle  $u \in L$  mit kleinstem Wert  $d[u]$   
        lösche  $u$  aus  $L$   
        Discovered[ $u$ ]  $\leftarrow$  true  
        foreach Kante  $e = (u, v) \in E$   
            if !Discovered[ $v$ ]  
                 $d[v] \leftarrow \min(d[v], d[u] + \ell_e)$ 
```

Heaps

Heapify-up

Heapify-up

Einfügen eines neuen Elements: Bei einem Heap mit n Elementen wird das neue Element an Position $n + 1$ eingefügt. Wir gehen dabei davon aus, dass noch genügend Plätze im Array frei sind.

Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element verletzt werden.

Reparieren: Durch Operation Heapify-up (für Heap-Array H an Position i) in $O(\log n)$ Zeit. Aufruf nach dem Einfügen des neuen Elements: $\text{Heapify-up}(H, n + 1)$.

```
Heapify-up(H, i):  
    if  $i > 1$   
         $j \leftarrow \lfloor i/2 \rfloor$   
        if  $H[i] < H[j]$   
            Vertausche die Array-Einträge  $H[i]$  und  $H[j]$   
            Heapify-up(H, j)
```

Heapify-down

Heapify-down

```
Heapify-down(H, i):  
    n ← length(H)-1  
    if  $2 \cdot i > n$   
        return  
    elseif  $2 \cdot i < n$   
        left ←  $2 \cdot i$ , right ←  $2 \cdot i + 1$   
        j ← Index des kleineren Wertes von H[left] und H[right]  
    else  
        j ←  $2 \cdot i$   
    if H[j] < H[i]  
        Vertausche die Arrayeinträge H[i] und H[j]  
        Heapify-down(H, j)
```

Erstellen eines Heaps:

[Erstellen eines Heaps](#)

[Erstellen](#): Das Erstellen eines Heaps aus einem Array A mit Größe n, das noch nicht die Heapeigenschaft erfüllt:

```
Init(A,n):  
  for i = ⌊n/2⌋ bis 1  
    Heapify-down(A,i)
```

Dijkstra-Algorithmus: Effizientere Variante

Dijkstra-Algorithmus: Effizientere Variante

Algorithmus:

- Arrays Discovered und d , Graph $G = (V, E)$, Startknoten s .
- Verwende Vorrangwarteschlange Q , in der die Knoten v nach dem Wert $d[v]$ geordnet sind.

```
Dijkstra( $G, s$ ):  
    Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
     $d[s] \leftarrow 0$   
     $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$   
     $Q \leftarrow V$   
    while  $Q$  ist nicht leer  
        wähle  $u \in Q$  mit kleinstem Wert  $d[u]$   
        lösche  $u$  aus  $Q$   
        Discovered[ $u$ ]  $\leftarrow$  true  
        foreach Kante  $e = (u, v) \in E$   
            if  $\text{!Discovered}[v]$   
                if  $d[v] > d[u] + \ell_e$   
                    lösche  $v$  aus  $Q$   
                     $d[v] \leftarrow d[u] + \ell_e$   
                    füge  $v$  zu  $Q$  hinzu
```

Greedy-Algorithmus

Geld wechseln

Geld wechseln: Greedy-Algorithmus

Greedy-Ansatz: Für Betrag S .

```
while  $S \neq 0$ 
    Finde die Münze mit größtem Wert  $x$ , sodass  $x \leq S$ 
    Benutze  $\lfloor S/x \rfloor$  Münzen von Wert  $x$ 
     $S \leftarrow S \bmod x$ 
```

Greedy-Ansatz konkreter:

- Werte von m Münzen in einem Array w .
- Es gilt $w[0] > w[1] > \dots > w[m - 1] = 1$.
- Betrag S gegeben.
- Anzahl jeder einzelnen Münze, um S zu wechseln, wird in einem Array num gespeichert.
- $num[i]$ enthält Anzahl der Münzen von Wert $w[i]$.

```
for  $i \leftarrow 0$  bis  $m - 1$ 
     $num[i] \leftarrow \lfloor \frac{S}{w[i]} \rfloor$ 
     $S \leftarrow S \bmod w[i]$ 
```

Interval Scheduling

Interval Scheduling: Greedy-Algorithmus

Greedy-Algorithmus: Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.

Wähle einen Job, wenn er kompatibel mit den bisher gewählten Jobs ist.

```
Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ 
 $A \leftarrow \emptyset$ 
for  $j \leftarrow 1$  bis  $n$ 
    if Job  $j$  ist kompatibel zu  $A$ 
         $A \leftarrow A \cup \{j\}$ 
return  $A$ 
```

■ Menge der ausgewählten Jobs

Greedy-Algorithmus: Pseudocode mit angepassten Indexwerten und Array.

```
Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ 
 $A \leftarrow \emptyset$ 
 $t \leftarrow 0$ 
for  $j \leftarrow 1$  bis  $n$ 
    if  $t \leq s_j$ 
         $A \leftarrow A \cup \{j\}$ 
         $t \leftarrow f_j$ 
return  $A$ 
```

Minimaler Spannbaum

Algorithmus von Prim

Algorithmus von Prim: Implementierung

Annahme: Alle Kantengewichte sind unterschiedlich.

```
Prim( $G, c$ ):  
    foreach ( $v \in V$ )  
         $A[v] \leftarrow \infty$   
    Initialisiere eine leere Priority Queue  $Q$   
    foreach ( $v \in V$ )  
        Füge  $v$  in  $Q$  ein  
     $S \leftarrow \emptyset$   
    while  $Q$  ist nicht leer  
         $u \leftarrow$  entnehme minimales Element aus  $Q$   
         $S \leftarrow S \cup \{u\}$   
        foreach Kante  $e = (u, v)$  inzident zu  $u$   
            if  $v \notin S$  und  $c_e < A[v]$   
                Verringere die Priorität  $A[v]$  auf  $c_e$ 
```

Algorithmus von Kruskal

Algorithmus von Kruskal: Implementierung

Implementierung:

```
Kruskal( $G, c$ ):  
Sortiere Kantengewichte so, dass  $c_1 \leq c_2 \leq \dots \leq c_m$   
 $T \leftarrow \emptyset$   
foreach ( $u \in V$ ) erzeuge eine einelementige Menge mit  $u$   
for  $i \leftarrow 1$  bis  $m$   
     $(u, v) = e_i$   
    if  $u$  und  $v$  sind in verschiedenen Mengen  
         $T \leftarrow T \cup \{e_i\}$   
        Vereinige die Mengen mit  $u$  und  $v$   
return  $T$ 
```

- *sind u und v in unterschiedlichen Zusammenhangskomponenten?*
- *Vereinige zwei Komponenten*

Union Find Datastruktur

Die Union Find Datenstruktur: Implementierung

Einfache Implementierung:

- makeset (v):

```
parent[v] = v
```

- union (v, w):

```
parent[v] = w
```

- findset (v):

```
h = v
while parent[h] ≠ h
    h = parent[h]
return h;
```

Laufzeit: Mit einer verbesserten Implementierung kann eine **in der Praxis nahezu konstante Laufzeit** für jede der drei Operationen erreicht werden.

Divide-and-Conquer

Mergesort

Aufteilen

Mergesort

Pseudocode:

- Mergesort für ein Array A.
- Sortiert den Bereich $A[l]$ bis $A[r]$.

```
Mergesort(A, l, r):  
    if l < r  
        m ← ⌊(l + r)/2⌋  
        Mergesort(A, l, m)  
        Mergesort(A, m + 1, r)  
        Merge(A, l, m, r)
```

Aufruf: $\text{Mergesort}(A, 0, n - 1)$ für ein Array A mit n Elementen.

Verschmelzen

Verschmelzen

Pseudocode: Merge auf ein Array A . Verwendet Hilfsarray B .

```
Merge(A,l,m,r):
    i ← l, j ← m + 1, k ← l
    while i ≤ m und j ≤ r
        if A[i] ≤ A[j]
            B[k] ← A[i], i ← i + 1
        else
            B[k] ← A[j], j ← j + 1
            k ← k + 1
        if i > m
            for h ← j bis r
                B[k] ← A[h], k ← k + 1
        else
            for h ← i bis m
                B[k] ← A[h], k ← k + 1
    for h ← l bis r
        A[h] ← B[h]
```

Quicksort

Quicksort

Quicksort: Benutzt auch das Divide-and-Conquer-Prinzip, aber auf eine andere Art und Weise.

Teile: Wähle „Pivotelement“ x aus Folge A ,
z.B. das an der letzten Stelle stehende Element.

Teile A ohne x so in zwei Teilfolgen A_1 und A_2 , dass gilt:

- A_1 enthält nur Elemente $\leq x$.
- A_2 enthält nur Elemente $\geq x$.

Herrsche:

- Rekursiver Aufruf für A_1 .
- Rekursiver Aufruf für A_2 .

Kombiniere: Bilde A durch Hintereinanderfügen von A_1 , x , A_2

Countsort

Lineare Sortierverfahren: Beispiel Countsort

Eingabe: n Zahlen im Bereich 0 bis z im Array A, Hilfsarray Counts, $z < n$

```
for j ← 0 bis z
    Counts[j] ← 0
for i ← 0 bis n - 1
    Counts[A[i]] ← Counts[A[i]] + 1
i ← 0
for j ← 0 bis z
    for k ← 0 bis Counts[j] - 1
        A[i] ← j
        i ← i + 1
```

Komplexität: Erste Schleife in $\Theta(z)$, zweite Schleife in $\Theta(n)$, dritte (mit vierter) Schleife kann nur maximal n Zahlen bearbeiten und ist daher auch in $\Theta(n)$. Damit läuft der Algorithmus in $\Theta(z + n + n) = \Theta(n)$ (da $z = O(n)$).

Inversionen zählen

Inversionen zählen: Implementierung

Precondition: [Merge-and-Count] A und B sind sortiert. **Postcondition:** [Sort-and-Count] L ist sortiert.

```
Sort-and-Count(L):
  if Liste L hat genau ein Element
    return 0 und die Liste L

  Unterteile die Liste in zwei Hälften A und B
  ( $r_A, A$ ) ← Sort-and-Count(A)
  ( $r_B, B$ ) ← Sort-and-Count(B)
  ( $r, L$ ) ← Merge-and-Count(A, B)

  return  $r_A + r_B + r$  und die sortierte Liste L
```

```
Merge-and-Count( $A, B$ ):
 $i \leftarrow 1, j \leftarrow 1$ 
 $count \leftarrow 0$ 
while beide Listen sind nicht leer
  if  $a_i \leq b_j$ 
    Füge  $a_i$  zur Ergebnisliste hinzu und erhöhe  $i$ 
  else
    Füge  $b_j$  zur Ergebnisliste hinzu und erhöhe  $j$ 
     $count \leftarrow count +$  die Anzahl der restlichen Elemente in  $A$ 
Füge den Rest der nicht leeren Liste zur Ergebnisliste hinzu
return  $count$  und sortierte Folge (Verschmelzung beider Hälften)
```

Dichteste Punktpaar

Dichtestes Punktpaar

Dichtestes Paar(p_1, \dots, p_n):

if $n = 1$ **return** ∞

Berechne Trennlinie L , sodass Punkte auf zwei Hälften aufgeteilt werden.

$O(n \log n)$

$\delta_1 = \text{Dichtestes Paar(linke Hälfte)}$

$2T(n/2)$

$\delta_2 = \text{Dichtestes Paar(rechte Hälfte)}$

$O(n)$

$\delta = \min(\delta_1, \delta_2)$

Lösche alle Punkte die weiter als δ von der Trennlinie L entfernt sind

Sortiere die restlichen Punkte nach y -Koordinate.

$O(n \log n)$

Scanne die Punkte in y -Reihenfolge und vergleiche die Distanz zwischen jedem Punkt und den nächsten 11 Nachbarn. Wenn eine dieser Distanzen kleiner als δ ist, ändere δ

$O(n)$

return δ .

Suchbäume

Inorder-Traversal

Durchmusterungen (Traversals): Inorder

Inorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.

Aufruf: $\text{Inorder}(root)$.

```
Inorder(p)
  if p ≠ null
    Inorder(p.left)
    Bearbeite p (z.B. Ausgabe)
    Inorder(p.right)
```

Laufzeit: Die Laufzeit liegt für $n \geq 1$ Knoten in $\Theta(n)$, da für jeden Knoten genau ein rekursiver Aufruf erfolgt, maximal $2n$ zusätzliche Aufrufe für leere Unterbäume hinzukommen, und jeder Aufruf ohne Folgeaufrufe eine Laufzeit von $\Theta(1)$ hat.

Preorder-Traversal

Durchmusterungen: Preorder

Preorder-Durchmusterung: Behandle rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum.

```
Preorder(p)
if p ≠ null
    Bearbeite p (z.B. Ausgabe)
    Preoder(p.left)
    Preorder(p.right)
```

Postorder-Traversal

Durchmusterungen: Postorder

Postorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel.

```
Postorder(p)
if p ≠ null
    Postorder(p.left)
    Postorder(p.right)
    Bearbeite p (z.B. Ausgabe)
```

Suche Knoten mit Wert S im Suchbaum

Suchen

Eingabe: Baum mit Wurzel p und gesuchter Schlüssel s

Rückgabewert: Knoten mit Schlüssel s oder $null$, falls s nicht vorhanden ist.

```
Search( $p, s$ ):  
  while  $p \neq null$  und  $p.key \neq s$   
    if  $s < p.key$   
       $p \leftarrow p.left$   
    else  
       $p \leftarrow p.right$   
  return  $p$ 
```

Suche Knoten mit kleinstem Wert im Suchbaum

Minimum

Eingabe: Baum mit Wurzel p .

Rückgabewert: Knoten mit dem kleinsten Schlüssel.

```
Minimum( $p$ ):  
    if  $p = \text{null}$   
        return  $\text{null}$   
    while  $p.\text{left} \neq \text{null}$   
         $p \leftarrow p.\text{left}$   
    return  $p$ 
```

Vorgehen: Solange beim linken Kind weitergehen, bis es keinen linken Nachfolger mehr gibt.

Suche Knoten mit größtem Wert im Suchbaum

Maximum

Eingabe: Baum mit Wurzel p .

Rückgabewert: Knoten mit dem größten Schlüssel.

```
Maximum( $p$ ):  
    if  $p = \text{null}$   
        return  $\text{null}$   
    while  $p.\text{right} \neq \text{null}$   
         $p \leftarrow p.\text{right}$   
    return  $p$ 
```

Vorgehen: Solange beim rechten Kind weitergehen, bis es keinen rechten Nachfolger mehr gibt.

Successor

Successor

Eingabe: Knoten p in Baum.

Rückgabewert: Nächster Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Successor( $p$ ):  
    if  $p.right \neq null$   
        return Minimum( $p.right$ )  
    else  
         $q \leftarrow p.parent$   
        while  $q \neq null$  und  $p = q.right$   
             $p \leftarrow q$   
             $q \leftarrow q.parent$   
        return  $q$ 
```

Predecessor

Predecessor

Eingabe: Knoten p in Baum.

Rückgabewert: Vorhergehender Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Predecessor( $p$ ):  
    if  $p.left \neq null$   
        return Maximum( $p.left$ )  
    else  
         $q \leftarrow p.parent$   
        while  $q \neq null$  und  $p = q.left$   
             $p \leftarrow q$   
             $q \leftarrow q.parent$   
        return  $q$ 
```

Einfügen

Einfügen

Eingabe: Baum mit Wurzel $root$ und ein neuer Knoten q .

Hinweis: $root$ wird bei leerem Baum verändert (Referenzparameter)

```
Insert(root, q):
r ← null, p ← root
while p ≠ null
    r ← p
    if q.key < p.key
        p ← p.left
    else
        p ← p.right
    q.parent ← r, q.left ← null, q.right ← null
    if r = null
        root ← q
    else
        if q.key < r.key
            r.left ← q
        else
            r.right ← q
```

Entfernen eines Knoten

Entfernen eines Knotens

Eingabe: Baum mit Wurzel $root$ und ein Knoten q .

Hinweis: $root$ kann verändert werden (Referenzparameter)

```
Remove(root, q):
    if q.left = null oder q.right = null
        r ← q
    else
        r ← Successor(q), q.key ← r.key, q.info ← r.info
        if r.left ≠ null
            p ← r.left
        else
            p ← r.right
        if p ≠ null
            p.parent ← r.parent
        if r.parent = null
            root ← p
        else
            if r = r.parent.left
                r.parent.left ← p
            else
                r.parent.right ← p
```

AVL-Bäume

Höhe

Höhe

Implementierung: Wir ergänzen alle Knoten u um ein Attribut $u.height$, das jeweils die Höhe des (Unter-)baumes mit der Wurzel u angibt.

Hilfsfunktion: Wir definieren folgende Hilfsfunktion zur Ermittlung der Höhe eines Baumes:

- Eingabe: Teilbaum mit Wurzel u .
- Rückgabewert: Höhe des Teilbaums.

```
Height( $u$ ):  
  if  $u = null$   
    return -1  
  else  
    return  $u.height$ 
```

Einfache Rotation nach rechts

Einfache Rotation nach rechts

Eingabe: Teilbaum mit Wurzel u .

Rückgabewert: Neue Wurzel v des rebalancierten Teilbaums.

```
RotateToRight( $u$ ):  
     $v \leftarrow u.left$   
     $u.left \leftarrow v.right$   
     $v.right \leftarrow u$   
     $u.height \leftarrow \max(\text{Height}(u.left), \text{Height}(u.right)) + 1$   
     $v.height \leftarrow \max(\text{Height}(v.left), \text{Height}(u)) + 1;$   
    return  $v$ 
```

Doppelrotation links-rechts

Doppelrotation links-rechts

Eingabe: Teilbaum mit Wurzel u .

Rückgabewert: Neue Wurzel des rebalancierten Teilbaums.

```
DoubleRotateLeftRight( $u$ ):  
     $u.left \leftarrow \text{RotateToLeft}(u.left)$   
    return  $\text{RotateToRight}(u)$ 
```

Einfügen in einen AVL-Bäume

Einfügen in einen AVL-Baum

Eingabe: Wurzel p (wird mögl. geändert), neuer Knoten q .

Rückgabewert: Höhe des Teilbaums.

```
Insert( $p, q$ ):  
    if  $p = null$   
         $p \leftarrow q, q.left \leftarrow q.right \leftarrow null, q.height \leftarrow 0$   
    else  
        if  $q.key < p.key$   
            ...  
        elseif  $q.key > p.key$   
            Insert( $p.right, q$ )  
            if Height( $p.right$ ) - Height( $p.left$ ) = 2  
                if Height( $p.right.right$ ) ≥ Height( $p.right.left$ )  
                     $p \leftarrow \text{RotateToLeft}(p)$   
                else  
                     $p \leftarrow \text{DoubleRotateRightLeft}(p)$   
            else  
                Knoten  $q$  schon vorhanden  
 $p.height \leftarrow \max(\text{Height}(p.left), \text{Height}(p.right)) + 1$ 
```

B-Bäume und B*-Bäume

Suche

Suche

Eingabe: B-Baum mit Wurzel p und ein Schlüssel x .

Rückgabewert: Knoten mit Schlüssel x oder $null$, falls x nicht vorhanden ist.

```
Search( $p, x$ ):  
     $i \leftarrow 1$   
    while  $i \leq p.l$  und  $x > p.key[i]$   
         $i \leftarrow i + 1$   
    if  $i \leq p.l$  und  $x = p.key[i]$   
        return ( $p, i$ )  
    if  $p.l = 0$   
        return  $null$   
    else  
        return Search( $p.child[i - 1], x$ )
```

Hashing

Initialisierung

Initialisierung

Eingabe: Hashtabelle $T = \text{Array von } m \text{ Verweisen auf die jeweils ersten Elemente.}$

Ergebnis: Initialisierte Hashtabelle.

```
Initialize(T, m):
  for i ← 0 bis m − 1
    T[i] = null
```

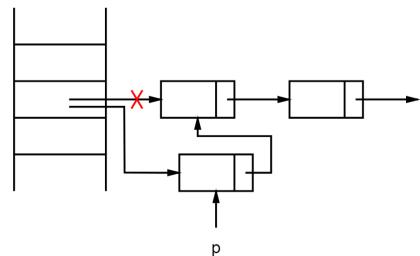
Einfügen

Einfügen

Eingabe: Hashtabelle T und einzufügendes Element p .

Ergebnis: Hashtabelle T mit neu eingetragenem Element p .

```
Insert(T, p):  
pos ← h(p.key)  
p.next ← T[pos]  
T[pos] ← p
```



Hinweis: Der Hashwert (und damit die Position in der Hashtabelle) für $p.key$ wird mit der Funktion h berechnet.

Suchen

Suchen

Eingabe: Hashtabelle T und gesuchter Schlüssel k.

Rückgabewert: Gesuchtes Element p.

```
Search(T, k):
    p = T[h(k)]
    while p ≠ null und p.key ≠ k
        p ← p.next
    return p
```

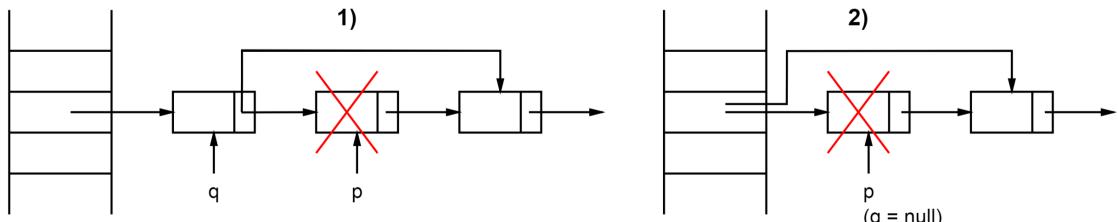
Entfernen eines Elements

Entfernen eines Elements

Eingabe: Hashtabelle T und Schlüssel k des zu entfernenden Elements (wir gehen davon aus, dass ein Element mit dem gesuchten Schlüssel in T enthalten ist).

Ergebnis: Element mit dem Schlüssel k wurde aus T entfernt.

```
Remove(T, k):  
    pos ← h(k)  
    q ← null  
    p ← T[pos]  
    while p.key ≠ k  
        q ← p  
        p ← p.next  
    if q = null  
        T[pos] ← T[pos].next  
    else  
        q.next ← p.next;
```



Einfügen nach Brent

Einfügen nach Brent

Eingabe: Hashtabelle T und neuer Schlüssel k .

```
Insert-Brent( $T, k$ ):  
     $j \leftarrow h_1(k)$   
    while  $T[j].status = used$   
         $k' \leftarrow T[j].key$   
         $j_1 \leftarrow (j + h_2(k)) \bmod m$   
         $j_2 \leftarrow (j + h_2(k')) \bmod m$   
        if  $T[j_1].status \neq used$  oder  $T[j_2].status = used$   
             $j \leftarrow j_1$   
        else  
             $T[j] \leftarrow k$   
             $k \leftarrow k'$   
             $j \leftarrow j_2$   
     $T[j] \leftarrow k$   
     $T[j].status \leftarrow used$ 
```

Praktische Datenstrukturen in Java - Ein Überblick

Timsort

Timsort

Grundlegende Idee: Finde schon sortierte Stücke (Runs) des Inputs S , die dann paarweise verschmelzt werden (Merging).

```
Timsort( $S$ )
runs  $\leftarrow$  partitioniere  $S$  in Runs
 $\mathcal{R} \leftarrow$  leerer Stack
while runs  $\neq \emptyset$ 
    entferne Run  $r$  von runs und pushe  $r$  auf den Stack  $\mathcal{R}$ 
    merge_collapse( $\mathcal{R}$ ) // ausbalanciertes Verschmelzen
while height( $\mathcal{R}$ )  $\neq 1$ 
    verschmelze die beiden oberen Runs am Stack
```

Timsort: Invarianten und merge_collapse

Stack \mathcal{R} von Runs mit Aufbau $\mathcal{R}[1], \dots, \mathcal{R}[\text{height}(\mathcal{R})]$, wobei $\mathcal{R}[1]$ top-of-stack ist. Die Länge von Run $\mathcal{R}[i]$ wird mit r_i bezeichnet.

```
merge_collapse( $\mathcal{R}$ )
while height( $\mathcal{R}$ )  $> 1$ 
    if height( $\mathcal{R}$ )  $> 2$  und  $r_3 \leq r_2 + r_1$ 
        if  $r_3 < r_1$ 
            verschmelze  $\mathcal{R}[2]$  und  $\mathcal{R}[3]$  am Stack
        else
            verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
    elseif height( $\mathcal{R}$ )  $> 1$  und  $r_2 \leq r_1$ 
        verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
    else break
```

Nach Ausführung von $\text{merge_collapse}(\mathcal{R})$ gelten folgende Invarianten:

- (1) $r_3 > r_2 + r_1$
- (2) $r_2 > r_1$