Algorithmen und Datenstrukturen

Ano Nym & Reiner Zafall

SOSE 2022

Contents

1 Sortieralgorithmen

- Sequenz von n Elementen $\{x_0, x_1, x_2, ..\}$
- Ordnungsrelation auf den Elementen
  - Vergleichsoperator auf 2 Elemente
- gemäß der Vergleichsordnung ($\leq$) sortiert

Vergleichsbasierte Sortieralgorithmen können im "Worstcase" mindestens $O(n \cdot \log n)$ erreichen. Da es bei n Einträge $n!$ mögliche Vergleiche gibt (true/false) hat der binäre Baum eine Tiefe von $\log_2(n!) \approx n \cdot \log(n)$

Korrektheit

Korrektheit mathematischer Beweis, dass die Ausgabe stimmt und der Algorithmus terminiert. Dies erfolgt üblicher Weise durch Induktion.
Dazu sind folgende Fälle in der **Induktionsverankerung ($i = 0$)** und dem **Induktionsschritt ($i > 0$)** zu beachten:

(a) A enthält die gleichen Werte wie am Anfang

(b) $A[0..i]$ ist korrekt sortiert

(c) "Werte in $A[0..i]$" $\leq$ "Werte in $A[i+1..n-1]$"

Laufzeit

Die Laufzeit gibt an, wie sich die Anzahl der Elemente in der Sequenz auf die Geschwindigkeit (Zeit bis zur Terminierung) des Algorithmus ausübt.
Man betrachtet die Grundoperationen, da diese etwa alle gleich lange benötigen und man so ein relatives Maß, unabhängig von Compiler und Sprache erhält. Man betrachtet hier besonders Schleifendurchläufe.
So hat eine "for-loop" lineare Auswirkung. 2 Loops sind somit quadratisch.
Hierfür benutzt man die "O-Notation"

Landau-Symbole

- $O(g(n)) := \{f(n) | \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
  - Funktion $f(n) \in O(g(n))$, falls es Konstanten c und n_0 gibt, sodass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$
  - $f(n) \leq g(n)$, asymptotisch gesehen
  - $f(n)$ wächst asymptotisch nicht schneller als $g(n)$
  - $\lim n \to \infty \frac{f(n)}{g(n)} < \infty$
- $\Omega(g(n)) := \{f(n) | \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  - Funktion $f(n) \in \Omega(g(n))$, falls es Konstanten c und n_0 gibt, sodass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_0$
  - $f(n) \geq g(n)$, asymptotisch gesehen
  - $f(n)$ wächst asymptotisch mindestens so schnell, wie $g(n)$
  - $\lim n \to \infty \frac{f(n)}{g(n)} > \infty$
- $\Theta(g(n)) := O(g(n)) \cap \Omega(g(n))$
  - Funktion $f(n) \in \Omega(g(n))$, falls es Konstanten c_1, c_2 und n_0 gibt, sodass $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ für alle $n \geq n_0$, resp. falls $f(n) \in O(n)$ und $f(n) \in \Omega(n)$
  - $f(n) = g(n)$, asymptotisch gesehen
  - $f(n)$ wächst asymptotisch gleich schnell, wie $g(n)$
  - $0 < \lim n \to \infty \frac{f(n)}{g(n)} < \infty$
- $o(g(n)) := \{f(n) | \forall c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$

- Funktion $f(n) \in o(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \leq c \cdot g(n)$ (für genug große n, abhängig von c)
- $f(n) < g(n)$, asymptotisch gesehen
- $f(n)$ wächst asymptotisch langsamer als $g(n)$
- $\lim n \to \infty \frac{f(n)}{g(n)} = 0$

- $\omega(g(n)) := \{f(n) | \forall c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
  - Funktion $f(n) \in \omega(g(n))$, falls für alle Konstanten $c > 0$ gilt, dass $f(n) \geq c \cdot g(n)$ (für genug große n, abhängig von c)
  - $f(n) > g(n)$, asymptotisch gesehen...
  - $f(n)$ wächst asymptotisch schneller als $g(n)$
  - $\lim n \to \infty \frac{f(n)}{g(n)} = \infty$

1.1 Selection Sort

1. Finde kleinstes Element im Array, vertausche es an 1. Stelle
2. Finde kleinstes Element im Rest, vertausche es an 2. Stelle
3. Finde kleinstes Element im Rest, vertausche es an 3. Stelle

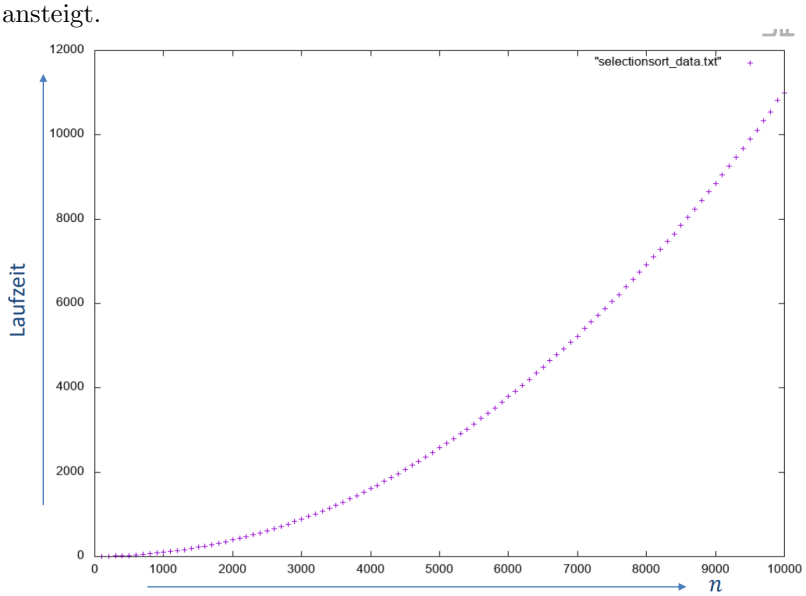
Stabil

Pseudocode

```
1   def SelectionSort(A):
2       for i=0 to n-2 do
3       // find min in A[i..n-1]
4           minIdx = i
5           for j=i+1 to n-1 do
6               if A[j] < A[minIdx] then
7                   minIdx = j
8           // swap A[i] with min of A[i..n-1]
9           tmp = A[i]
10          A[i] = A[minIdx]
11          A[minIdx] = tmp
```

Analyse

Durch einfache Zeitmessung des Codes, lässt sich schnell erkennen, dass die benötigte Zeit unverhältnismäßig ansteigt.



Man kann erkennen, dass die Laufzeit für Selectionsort $O(n^2)$ beträgt

1.2 Insertion Sort

1. Erstes Element gilt als sortiert
2. Nächste Element wird je nachdem ob es größer ist, vor oder hinter das erste Element gelegt
3. Das nächste wird je nach Größe vor oder hinter die beiden Elemente gelegt

Instabiel

Pseudocode

```
1       def InsertionSort(A):
2           for i=0 to n-2 do
3               // prefix A[0..i] is already sorted
4               pos = i+1
5               while (pos > 0) and (A[pos] < A[pos-1]) do
6                   swap(A[pos], A[pos-1])
7                   pos = pos - 1
```

Analyse

Durch die zwei Schleifen ist zu erkennen, dass die Laufzeit $O(n^2)$ beträgt.

1.3 QuickSort

1. Man wählt ein Pivotelement (meistens der Median)
2. das Array wird in 2 Arrays aufgeteilt.
   - Alle Elemente kleiner als das Pivot kommen in das Linke Array
   - Alle Elemente größer oder gleich kommen in das rechte Array
3. Dieser Schritt wird für die so entstandenen Arrays rekursiv wiederholt, bis die Zwischenarrays die Größe 2 haben

Um Speicher zu sparen, kann man die Unterteilung des Arrays auch mit Pointern Darstellen!

1. Zwei Variablen l und r, um von links und rechts durchs Array zu gehen
2. Inkrementiere l bis $A[l] > x$ (Element muss nach rechts)
3. Dekrementiere r bis $A[l] \leq x$ (Element muss nach links)
4. Vertausche, und stelle l und r eins vor ($l+=, r-=1$)
5. Divide fertig, sobald sich l und r treffen

Instabiel

Pseudocode

```
# Python program for implementation of Quicksort Sort

# This implementation utilizes pivot as the
# last element in the nums list
# It has a pointer to keep track of the elements smaller than the pivot
# At the very end of partition() function,
# the pointer is swapped with the pivot
# to come up with a "sorted" nums relative to the pivot


def partition(l, r, nums):
    # Last element will be the pivot and the first element the pointer
    pivot, ptr = nums[r], l
    for i in range(l, r):
        if nums[i] <= pivot:
            # Swapping values smaller than the pivot to the front
            nums[i], nums[ptr] = nums[ptr], nums[i]
            ptr += 1
    # Finally swapping the last element with the pointer indexed number
    nums[ptr], nums[r] = nums[r], nums[ptr]
    return ptr

# With quicksort() function, we will be utilizing
# the above code to obtain the pointer
# at which the left values are all smaller than
# the number at pointer index and vice versa
# for the right values.


def quicksort(l, r, nums):
    # Terminating Condition for recursion. VERY IMPORTANT!
    if len(nums) == 1:
        return nums
    if l < r:
        pi = partition(l, r, nums)
        quicksort(l, pi-1, nums) # Recursively sorting the left values
        quicksort(pi+1, r, nums) # Recursively sorting the right values
    return nums
```

Analyse

Je nach Wahl des Pivot kann QuickSort von $O(n \log n)$ bis $O(n^2)$.
Es ist zu erkennen, dass das Pivot entscheidend ist.
Meist am schnellsten mit random QuickSort

1.4 MergeSort

1. Man teilt das Array in Subarrays der Größe 1 auf
2. Durch Pointer werden zwei Elemente verglichen und das kleinere wird in ein neues Array eingesetzt
3.

Instabiel

Pseudocode

```
1     def MergeSort(A):
2         allocate array tmp to store intermediate results
3         MergeSortRecursive(A, 0, n, tmp)
4
5     def MergeSortRecursive(A, start, end, tmp):
6         if end - start > 1:
7             # integer division
8             middle = start + (end - start) / 2
9             MergeSortRecursive(A, start, middle, tmp)
10            MergeSortRecursive(A, middle, end, tmp)
11            pos = start; i = start; j = middle
12            while pos < end:
13                if i < middle and (j >= end or A[i] <= A[j]):
14                    tmp[pos] = A[i]; pos++; i++
15                else:
16                    tmp[pos] = A[j]; pos++; j++
17            for i = start to end-1 do A[i] = tmp[i]
```

Analyse

$O(n * \log n)$ - fast linear, in jedem Fall

1.5 Counting Sort

1. Zählt wie oft jedes Element vorkommen und speichert diese in einem Array
2. Die Länge/Größe dieses Arrays beträgt die Größe des Intervalls der Elemente in der zu sortierenden Sequenz
3. Dieses Array gibt die Position im sortierten Zustand an
4. Dabei wird der Wert in Zählerarray um 1 verringert

Stabil und nicht Vergleichs basiert

Pseudocode

```
1     counts = new int[k+1] // new int array of length k
2     for i = 0 to k do counts[i] = 0
3     for i = 0 to n-1 do counts[A[i]]++
4     i = 0;
5     for j = 0 to k do
6         for l = 1 to counts[j] do
7             A[i] = j; i++
```

Analyse

$O(n + k)$ mit k größte Zahl im Array (Größe des Zahlenintervalls)

1.6 Heapsort

```
1     H = new BinaryHeap()
2     for i = 0 to n - 1 do
3         H.insert(A[i])
4     for i = 0 to n - 1 do
5         A[i] = H.deleteMin()
```

Laufzeit: $O(n \log n)$

2 Datenstrukturen

2.1 Array

Einfach Kollektion von Elementen. Meisten im Speicher hintereinander angelegt.
Arrays haben eine feste Größe und sind sehr unflexibel

2.2 Dictionary

Implementierung mit Verkettete Listen

Implementierung

- Create: erzeugt einen leeren Dictionary
- D.insert(key, value): fügt neues (key,value)-Paar hinzu
- D.find(key): gibt Eintrag zu Schlüssel key zurück
- D.delete(key): : löscht Eintrag zu Schlüssel key

Laufzeit Vergleich der Laufzeit bei verschiedenen Implementierungen

	Verkettete Liste (unsortiert)	Array (unsortiert)	Array (sortiert)
insert	$O(1)$	$O(1)$	$O(n)$
delete	$O(n)$	$O(n)$	$O(n)$
find	$O(n)$	$O(n)$	$O(\log n)$

n: Aktuelle Anzahl Elemente im Dictionary

Efizientiste implementierung benutzt "Hashing"!!!

- create: $O(1)$
- insert: $O(1)$, falls man nicht überprüfen muss, ob der Schlüssel schon vorhanden ist
- find: $O(1)$ durch geeignete Struckturen
- delete: $O(1)$ da delete mit find verknüpft ist

Verwendung

- Wörterbuch (key: Wort, value: Definition / Übersetzung)
- Telefonbuch (key: Name, value: Telefonnummer)
- DNS-Server (key: URL, value: IP-Adresse)
- Python Interpreter (key: Variablenname, value: Wert der Variable)
- Java/C++ Compiler (key: Variablenname, value: Typinformation)

2.3 Queue

Array-Implementierung

```
1   S.isEmpty():
2       return (size == 0)
3   S.enqueue(x):
4       if (size < NMAX)
5           pos = (head + size) mod NMAX
6           ueue[pos] = x
7           size += 1
8   S.dequeue():
9       if (size == 0)
10          report error (or return default value)
11      else
12          x = queue[head]
13          head = (head + 1) mod NMAX
14          size = size - 1
15          return x
```

Laufzeit

- reate: $O(1)$
- enqueue : $O(1)$
- dequeue : $O(1)$
- sEmpty : $O(1)$

Nachteile

- Speicherverbrauch (space complexity) : $\Theta(NMAX)$
- man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur NMAX Elemente aufnehme...
- Wir werden gleich sehen, wie man beides beheben kann...

Linked List

- Alle Operationen in $O(1)$ Zeit
- Grösse nicht beschränkt, Speicherverbrauch $O(n)$

2.4 Stack

Array-Implementierung

```
1   isEmpty():
2       return (size == 0)
3   S.push(x):
4       if (size < NMAX):
5           stack[size] = x
6           size += 1
7   S.pop():
8       if (size == 0):
9           report error (or return default value)
10      else:
11          size -= 1
12          return stack[size]
```

Laufzeit

- create: $O(1)$
- push: $O(1)$
- pop: $O(1)$
- isEmpty: $O(1)$

Nachteile

- Speicherverbrauch (space complexity) : $\Theta(NMAX)$
- man braucht immer gleich viel Speicher, egal wie viele Elemente im Stack gespeichert sind!
- Der Stack kann nur NMAX Elemente aufnehmen

Linked List

- Alle Operationen in $O(1)$ Zeit
- Grösse nicht beschränkt, Speicherverbrauch $O(n)$

Verwendung

- Umdrehen einer Sequenz
- Undo bei Editoren
- Programmstack für Funktionen/Methoden-Aufrufe

2.5 Verkettete Listen

Implementierung

Wennman als letztes Element ein "Sentinel" Element benutzt, fallen die Spezialfälle für Opperationen am Rand der Liste weg. Ein nachteil ist, dass bei solchen Listen der Speicherverbrauch erhöht wird.

```
1     class ListElement:
2         def __init__(self, key=0, data=None, next=None, prev=None):
3             self.key = key
4             self.data = data
5             self.next = next
6             self.prev = prev
7
8         def find(x):
9             current = first
10            while current != None and current.key != x:
11                current = current.next
12
13        def insert(x, y):  # add y after x
14            y.next = x
15            y.prev = x.prev
16            x.prev.next = y
17            x.prev = y
18
19        def remove(x):
20            x.prev.next = x.next
21            x.next.prev = x.prev
```

In C/C++ sind "next" und "prev" Pointer! Eine Linked-List besteht aus ListElementen, die untereinander auf sich zeigen.
Somit entsteht eine dynamische "Array" artige Strucktur.

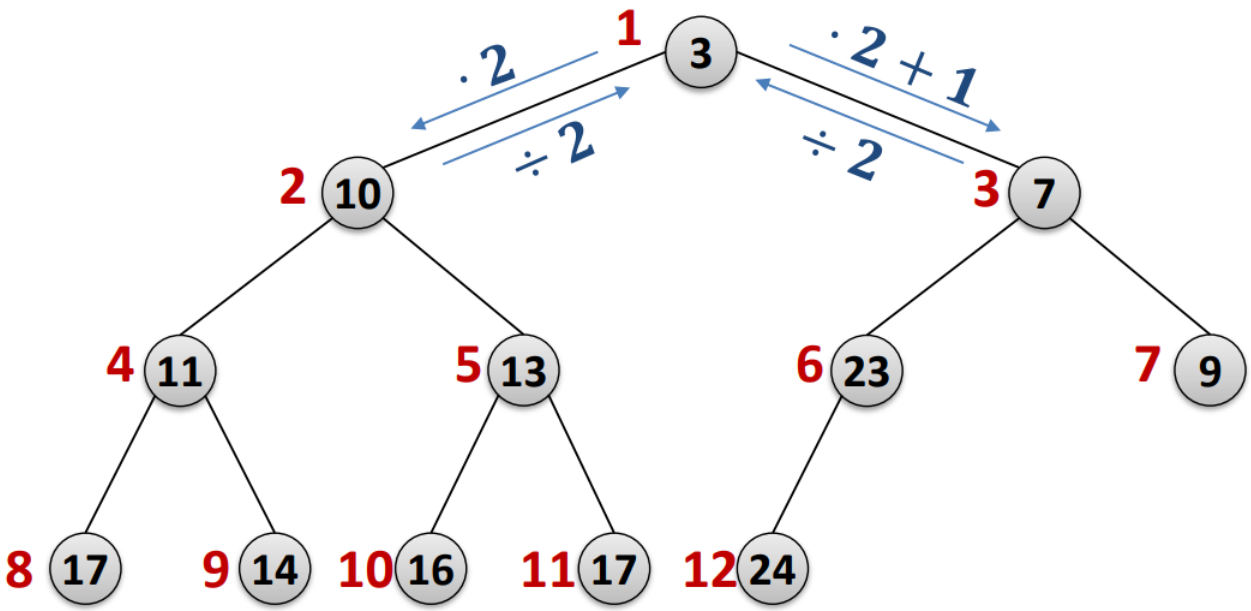
Laufzeit

- find: $O(n)$
- einfügen: $O(1)$
- löschen: $O(1)$
- concatenate: $O(1)$
-

2.6 Heap / Priority Queue

- Implementierung als Binärbaum
- Ein Baum hat die Min-Heap Eigenschaft, falls in jedem Teilbaum, die Wurzel den kleinsten Schlüssel hat
- Baum wird immer so balanciert, wie möglich gehalten
- So kann der Baum auch in einem Array gespeichert werden
- mithilfe von Arrays kann die Geschwindigkeit erhöht werden

Implementierung



Analyse

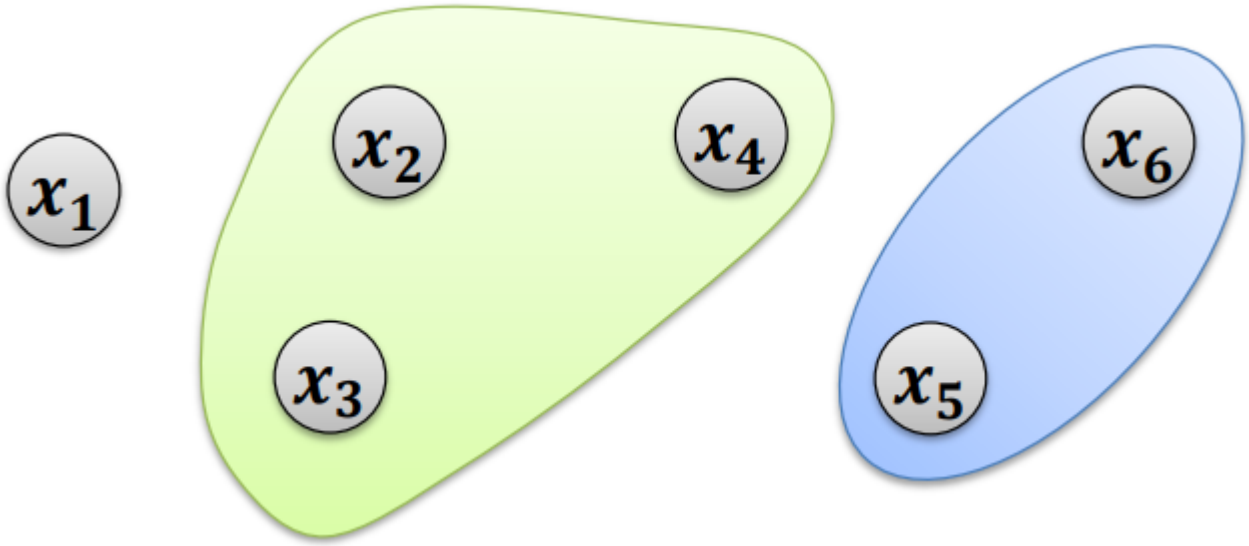
- Tiefe: $\lfloor \log_2 n \rfloor$
- Laufzeit: $O(\log n)$

2.7 Union-Find / Disjoint Sets

Diese Strucktur verwaltet eine Partition von Elementen

Operationen

- create() : erzeugt eine leere Union-Find-DS
- U.makeSet(x) : fügt Menge $\{x\}$ zur Partition hinzu
- U.find(x) : gibt Menge mit Element x zurück
- U.union(S1, S2) : vereinigt die Mengen S1 und S2



3 Hasching

3.1 Direkte Adressierung

Um eine belibig große Menge auf ein Array abzubileden benutzen wir Hashing.

- h (Hashfunktion) sollte möglichst zufällig sein
- Mapping verschiedener Schlüssel unabhängig
- Hashwerte sollten gleich oft vorkommen
- h sollte schnell berechnet werden

Da verschiedene Eingaben zu derselben Ausgabe führen kann es zu "Kollisionen" kommen.
Hierfür gibt es verschiedene Lösungsansätzte.

- Speichere beide (key,value)-Paare an die gleiche Stelle
- Speichere zweiten Schlüssel an eine andere Stelle

Hashtabellen mit Chaining

Jede Position im Array ist eine Verkettete Liste. bei einer Kollision wird der neue Schlüssel an dieselbe Stelle angehängt

Laufzeit

- create: $O(1)$
- create: $O(1)$ $O(1 + \text{Listenlänge})$ bei Kollisionen
- find: $O(1)$ $O(1 + \text{Listenlänge})$ bei Kollisionen
- delete: $O(1)$ $O(1 + \text{Listenlänge})$ bei Kollisionen

3.2 Lineares Sondieren

Bei einer Kollision wird das Element unetr Der Kollisionen eingefügt.
dadurch können sich aber Cluster bilden. Cluster der Grösse k wachsen in jedem Schritt mit Wahrscheinlichkeit $(k+2)/m$.

3.3 Quadratisches Sondieren

Ist ähnlich wie lineare Sondierung. Bei einer Kollision wird die Position genommen, welche nicht zu einem Kluster führt. Es können sich aber noch immer Cluster bilden.

3.4 Doppel-Hashing

$h(x,i) = h_1(x) + i \cdot h_2(x)) \mod m$

- Falls m eine Primzahl ist, werden alle Positionen abgedeckt
- Sondierungsfunktion hängt in zwei Arten von x ab
- Vermeidet die Nachteile von linearem und quadr. Sondieren
- sehr geringe warscheinlichkeit $\frac{1}{m^2}$ für eine Kollision

3.5 Universelles Hashing

- Wenn man die Hashfunktion zufällig aus einer universellen Menge von Hashfunktionen wählt, ist die Kollisionswahrscheinlichkeit für zwei Schlüssel x und y gleich, wie bei einer zufälligen Fkt.
- Es gibt einfache und effiziente Konstruktionen von universellen Mengen von Hashfunktionen

3.6 Rehash

- $\alpha = n/m$
- Nötig, wenn Hashtabelle voll oder schlecht verteilt
- Auswahl einer neuen Haschfunktion und vergrößerung des Arrays (ca. verdoppelt)
- zusätzliche Kosten $\Theta(m+n)$
- Sollte wenn möglich vermieden werden

3.7 Cuckoo Hashing

- Offene Adressierung
- Zwei Hashfunktionen h_1 und h_2
- Ein Schlüssel x wird immer bei $h_1(x)$ oder $h_2(x)$ gespeichert
- Falls ein anderer Schlüssel y an der Stelle $h_1(x)$:
  - Werfe y da raus (daher der Name: Cuckoo Hashing)
  - falls es bei $h_1(y)$ war, an Stelle $h_2(y)$
  - falls da auch schon ein Element z ist, werfe z raus und platziere es an seiner Alternativposition
- Find/Delete sind $O(1)$
- Zyklen können sich bilden →Rehash

4 Suchbäume

- linkes Element kleiner, rechtes größer als der Parent
- Tiefe zwischen $\log_2 n$ und $n-1$
- Alle Opperationen sind $O(\log n)$

4.1 Traversierung

- Tiefensuche
  - In-Order: In aufsteigender Reihenfolge
  - Pre-Order: Parents vor Kindern mit voller Tiefe
  - Post-Order: Kinder, dann Parents auslesen
- Breitensuche
  - Level-Order: Die einzelnen Ebenen von oben nach unten durch

Algorithmen der Tiefensuche konnen rekursiv einfach implementiert werden.

```
preorder(node):
    if node is not None:
        visit(node)
        preorder(node.left)
        preorder(node.right)

inorder(node):
    if node is not None:
        inorder(node.left)
        visit(node)
        inorder(node.right)

postorder(node):
    if node is not None:
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

BFS-Algorithmen lassen sich nicht so einfach rekursiv implementieren:

```
BFS–Traversal:
Q = new Queue()
Q.enqueue(root)
while not Q.empty():
    node = Q.dequeue()
    visit(node)
    if node.left is not None:
        Q.enqueue(node.left)
    if node.right is not None:
        Q.enqueue(node.right)
```

Laufzeit

- Tiefensuche:
  - Alle Knoten weden einmal besucht
  - $O(n)$
- Breitensuche:
  - Alle Knoten weden einmal besucht
  - $O(n)$

Anwendung

- In-Order: sortierenden
- Pre-Order: Geeignet um den Baum zu speichern
- Post-Order: löschen eines Baumes um Speicher freizumachen

4.2 Binäre Suche (Array)

Das Array oder die Liste muss sortiert vorliegen

1. testen, ob das Gesuchte Element größer oder kleiner ist als das mittlere Element im Array
2. Der teil (links oder rechts), in dem sich das gesuchte Element befindet wird behalten
3. Diese Schritte weden wiederholt, bis das Element gefunden wurde, oder die Teilgröße des Arrays 1 beträgt

Implementierung

```
l = 0; r = n - 1;
    while r > l do
        m = (l + r) / 2;
        if A[m] < x then
            l = m + 1
        else if A[m] > x then
            r = m - 1
        else
            l = m; r = m
```

Analyse

$O(\log n)$

4.3 Binäre Suche (Baum)

Implementierung

```
current = root
while current is not None and current.key != x:
    if current.key > x:
        current = current.left
    else:
        current = current.right
```

Analyse

O(Tiefe des Baumes)

4.4 Heap (Min-Heap)

- Wurzel jedes Teilbaumes ist das kleinste Element
- Root ist also das kleinste Element in der Strucktur

4.5 Treap

- Jedes Element hat zwei Schlüssel key1 und key2
- BST bezüglich key1
- Min-Heap bezüglich key2
- Alle Opperationen sind O(Tiefe) also: $O(\log_2 n)$
- Wurde in der Vorlesung nicht detailiert besprochen

4.6 Rot-Schwarz-Bäume

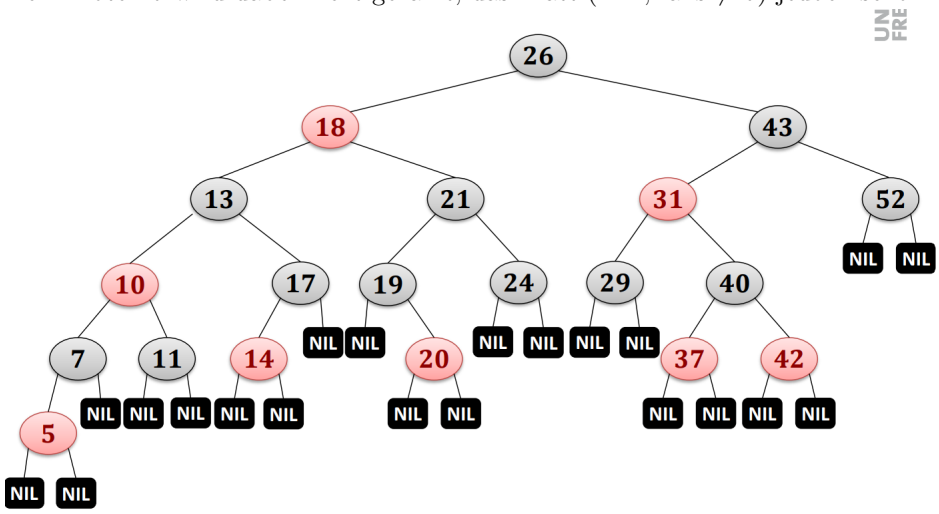
- Der RS-Baum ist balanciert
- Tiefe garantiert $\leq 2\log_2(n+1)$ und somit Laufzeit aller Opperationen $O(\log n)$

Tiefe

Die Tiefe(T) eines Knoten v ist die maximale Länge eines direkten Pfades im Teilbaum von v zu einem Blatt (NIL).

Schwarz-Tiefe

Die Schwarz-Tiefe(ST) eines Knoten v ist die Anzahl schwarzer Knoten auf jedem direkten Pfad von v zu einem Blatt (NIL).
Der Knoten v wird dabei nicht gezählt, das Blatt (NIL, falls $\neq v$) jedoch schon!



Eigenschaften

1. Alle Knoten sind rot oder schwarz
2. Wurzel ist schwarz
3. Blätter (= NIL-Knoten) sind schwarz
4. Rote Knoten haben zwei schwarze Kinder
5. Von jedem Knoten v haben alle direkten Pfade zu Blättern gleich viele schwarze Knoten →garantiert Balance
6. Tiefe(v) $\leq 2 \cdot$ Schwarz-Tiefe(v)
7. Die Anzahl innerer Knoten im Teilbaum eines Knoten $v (v \neq \text{NIL})$ mit Schwarz-Tiefe $ST(v)$ ist $\geq 2^{ST(v)} - 1$

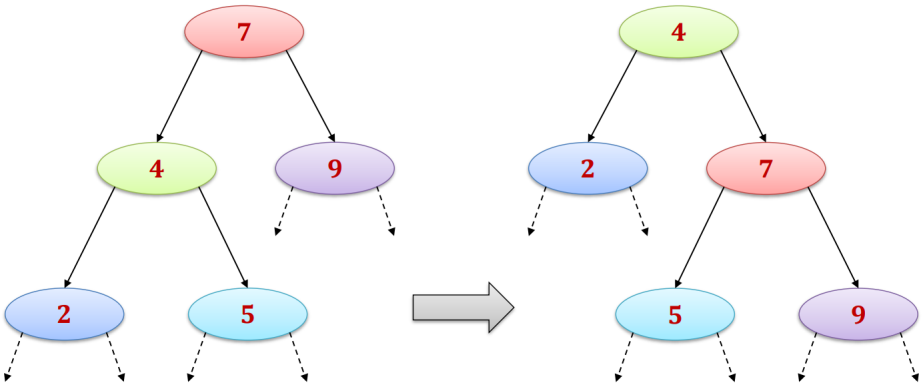
Einfügen

```
1     insert(x):
2         if root == NIL then
3             root = new Node(x,a,red,NIL,NIL,NIL)
4         else
5             v = root;
6             while v.key != x do
7                 if v.key > x then
8                     if v.left == NIL then
9                         w = new Node(x,a,red,v,NIL,NIL); v.left = w
10                    v = v.left
11                else if v.key < x then
12                    if v.right == NIL then
13                        w = new Node(x,a,red,v,NIL,NIL); v.right = w
14                    v = v.right
15            v.value = a
```

Rotation

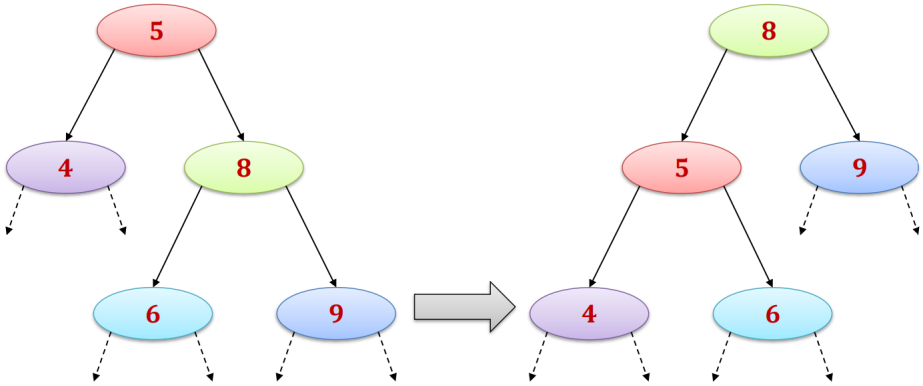
- Operation, um den binären Suchbaum lokal umzuordnen
- Ändert Topologie, erhält Binäre Suchbaum Eigenschaft
- Falls Root rot ist, einfach umfärben
- RS-Baum ist mithilfe von 2 Rotationen und einem Umfärben wieder korrekt
- Laufzeit $O(n)$

Rechtsrotation




```
right-rotate(u,v):
    u.left = v.right
    u.left.parent = u
    if u == root then
        root = v
    else
        if u == u.parent.left then
            u.parent.left = v
        else
            u.parent.right = v
    v.parent = u.parent
    v.right = u
    u.parent = v
```

Linksrotation




```
left-rotate(u,v):
    u.right = v.left
    u.right.parent = u
    if u == root then
        root = v
    else
        if u == u.parent.left then
            u.parent.left = v
        else
            u.parent.right = v
    v.parent = u.parent
    v.left = u
    u.parent = v
```

Löschen

- Wenn v rot ist, kann es einfach gelöscht weden
- Wenn v schwar ist und ein rotes Kind hat
  - lösche v
  - färbe das Kind um
- Wenn v nur NIL-Kinder hat, wird v gelöscht und der Baum korrigiert

4.7 AVL-Bäume

- Direkte Alternative zu Rot-Schwarz-Bäumen
- Binäre Suchäume bei denen gild: $|T(v.\text{left}) - T(v.\text{right})| \leq 1$
- Nutzt keine Farben, sondern merkt sich die Tiefe jedes Teilbaumes
- AVL-Bedingung kann bei insert/delete jeweils mit O(log n) Rotationen wieder hergestellt werden
- Suche ist etwas schneller, Einfügen/Löschen dafür etwas langsamer

4.8 (a, b)-Bäume

- Parameter $a \geq 2$ und $b \geq 2a - 1$
- Elemente/Schlüssel sind nur in den Blättern gespeichert
- Alle Blätter sind in der gleichen Tiefe
- Falls die Wurzel kein Blatt ist, hat sie zwischen 2 und b Kinder
- Alle anderen inneren Knoten haben zwischen a und b Kinder
- Ein a, b -Baum ist also kein Binärbaum!

4.9 B-Bäume

- Sclussel werden in den inneren Knoten gespeichert
- Hoher Speicherverbrauch
- Sind im Verhältnis zu BST recht flach
- Speziell gut z. B. für Dateisysteme (Zugriff sehr teuer)

4.10 AA-Trees

- ähnlich wie Rot-Schwarz-Bäume (nur rechte Kinder können rot sein)

4.11 Splay Trees

- Elemente, auf welche kürzlich zugegriffen wurde, sind weiter oben
- Gut, falls mehrere Knoten den gleichen Schlüssel haben können
- nicht streng balanciert

4.12 Skip Lists

- Verkettete Listen mit zusätzlichen Abkürzungen
- kein balancierter Suchbaum, hat aber ähnliche Eigenschaften

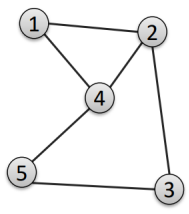
5 Graphen

- Graph $G = (V, E)$
- Knotenmenge V, typischerweise $n := |V|$
- Kantenmenge E, typischerweise $m := |E|$
- ungerichteter Graph $E \subseteq \{\{u, v\} : u, v \in V\}$
- gerichteter Graph $E \subseteq V \times V$

5.1 Repräsentation

- Adjazenzmatrix
  - Speichereffizient n^2 wenn, Graph nict zu dün besetzt
  - Wird für manche Algorithmen benötigt
- Adjazenzliste
  - Asymptotisch optimaler Speicherbedarf
  - Abfrage kann langsam sein
  - Gut für Tiefensuche

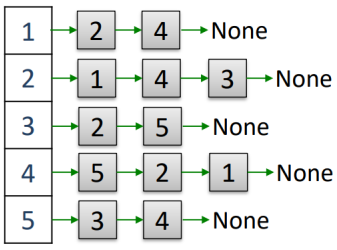
Beispiel:



Adjazenzmatrix

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

Adjazenzlisten



5.2 Traversierung

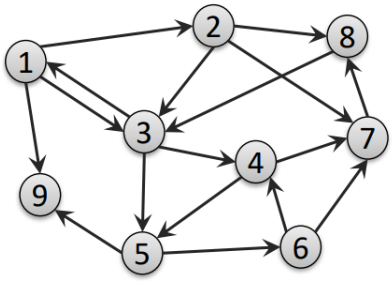
Da ein Graph im Gegensatzt zu einem Baum Zyklen enthalten kann, werden alle besuchten Knoten makiert, um nicht in einer Schleife gefangen zu werden.

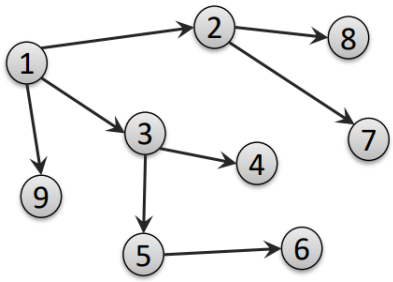
Breitensuche

Die Breitensuche ist nicht so einfach zu implementieren
Besucht zuerst alle Nachbarn von v. Wenn alle Nachbarn besucht wurden, werden dann erst alle Nachbern eines Nachbern besucht.
Die Laufzeit der BFS-Traversierung ist $O(n+m)$

```
BFS–Traversal(s):
    for all u in V: u.marked = false;
    Q = new Queue()
    s.marked = true
    Q.enqueue(s)
    while not Q.empty() do
        u = Q.dequeue()
        visit(u)
        for v in u.neighbors do
            if not v.marked then
                v.marked = true;
                Q.enqueue(v)
```

Wir vorallem benutzt, um einen BFS-Baum zu erstellen.





Tiefensuche

Es wird zuerst in die Tiefe gegangen, bevor man einen weiteren Nachbarn besucht. Wird rekursiv realisiert. DFS-Traversal entspricht der Postorder-Traversierung in Bäumen.
Fall man gleich beim Markieren den Knoten besucht, entspricht es der Preorder-Traversierun.
Die Laufzeit der Tiefensuche (DFS-Traversierung) ist $O(n+m)$.

```
1   DFS–Traversal(s):
2       for all u in V: u.color = white;
3       DFS–visit(s, NULL)
4   DFS–visit(u, p):
5       u.color = gray;
6       u.parent = p;
7       for all v in u.neighbors do
8           if v.color = white
9               DFS–visit(v, u)
10      visit node u;
11      u.color = black;
```

5.3 Zusammenhangskomponenten

Die Zusammenhangskomponenten (oder einfach Komponenten) eines Graphen sind seine zusammenhängenden Teile.

```
1    for u in V do
2        if not u.marked then
3            start new component
4            explore with DFS/BFS starting at u
```

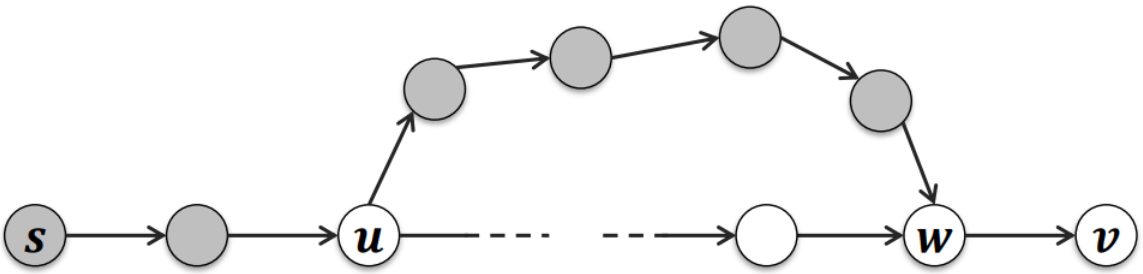
Man benötigt $O(n+m)$ Zeit um alle Komponenten zu identifiziern

5.4 DFS- "Klammer"-Theorem

- $t_{v,1}$: Zeitpunkt, wenn v in der DFS-Suche grau gefärbt wird
- $t_{v,2}$: Zeitpunkt, wenn v in der DFS-Suche schwarz gefärbt wird
- **Theorem**: Im DFS-Baum ist ein Knoten v genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.
- In Worten: Wenn v nach u grau, und vor u schwarz gefärbt wird ist es in seinem Teilbaum.
- Insbesondere sind die Intervalle entweder disjunkt oder komplett im anderen enthalten.

5.5 Weiße Pfade

Theorem: In einem DFS-Baum ist ein Knoten v genau dann im Teilbaum eines Knotens u, falls unmittelbar vor dem Markieren von u, ein komplett weißer Pfad von u nach v besteht.



5.6 Klassifizierung der Kanten (bei DFS-Suche)

Baumkanten:

- (u, v) ist eine Baumkante, falls v von u aus entdeckt wird
  - Bei Betrachen von (u, v) ist v weiß

Rückwärtskanten:

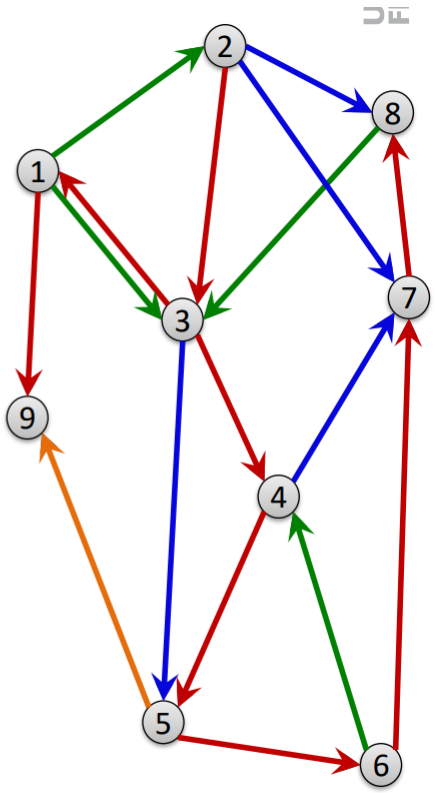
- (u, v) ist eine Rückwärtskante, falls v ein Vorgängerknoten von u ist
  - Bei Betrachten von (u, v) is v grau

Vorwärtskanten:

- (u, v) ist eine Vorwärtskante, falls v ein Nachfolgerknoten von u ist
  - Bei Betrachten von (u, v) ist v schwarz

Querkanten:

- Alle übrigen Kanten
  - Bei Betrachten von (u, v) ist v schwarz



5.7 DFS – Ungerichtete Graphen

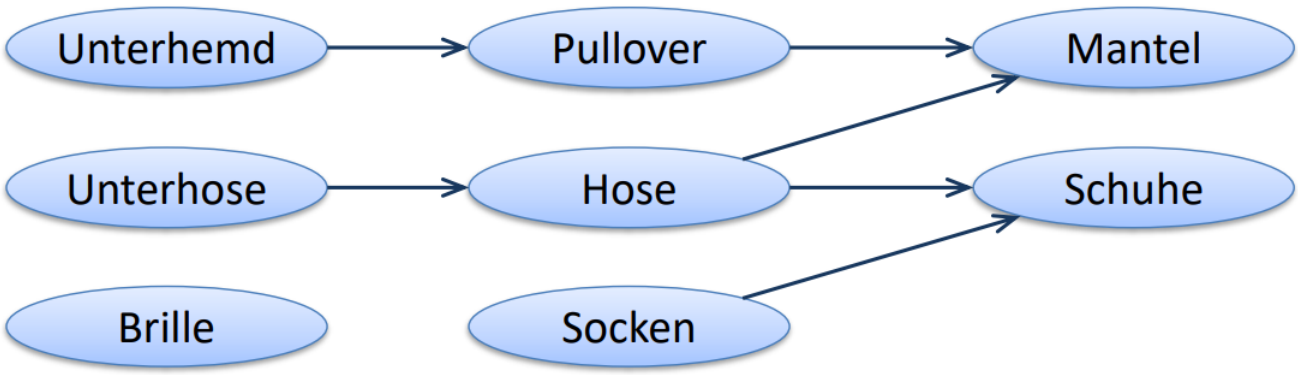
Theorem: Bei einer DFS-Suche in ungerichteten Graphen ist jede Kante entweder eine Baumkante oder eine Rückwärtskante

5.8 DFS – Gerichtete Graphen

Theorem: Ein gerichteter Graph hat genau dann keine Zyklen, falls es bei der DFS-Traversierung keine Rückwärtskanten gibt.

5.9 Topologische Sortierung

- Zyklenfreie, gerichtete Graphen
- Modellieren z. B. zeitliche Abhängigkeiten von Aufgaben



Theorem: Umgekehrte "Visit"-Reihenfolge (schwarz färben) der Knoten bei DFS-Traversierung ergibt topologische Sortierung eines gerichteten zyklenfreien Graphen.

5.10 DFS-Traversierung: Weitere Anwendung

- Stark zus.-hängende Komponente eines gerichteten Graphen: "Maximale Knoten-Teilmenge, sodass jeder jeden erreichen kann"
- Benötigt 2 DFS-Traversierungen (Zeit $\in O(m+n)$)
  - auf G und auf G^T (alle Kanten umgedreht)
  - G und G^T haben die gleichen stark zus.-hängenden Komponenten

5.11 Minimaler Spannbaum

- Minimaler Spannbaum = Minimum Spanning Tree = MST
- Spannbaum mit kleinstem Gesamtgewicht
- Ein grundlegendes Optimierungsproblem auf Graphen
- **Theorem:** Sei $(S, V/S)$ ein Schnitt, sodass A keine Schnittkanten enthält und sei $\{u,v\}, u \in S, v \in V/S$ eine leichte Schnittkante bezüglich $(S, V/S)$. Dann ist $\{u,v\}$ eine sichere Kante für A.

5.12 Prims MST Algorithmus

Zählt zu den Greedy Algorithmen

```
1    A = 0
2    while A ist kein Spannbaum do
3        Finde sichere Kante u, v -> A
4        A = A or {{u, v}}
5    return A
```

- Gehe von einem Anfangsknoten aus und füge immer die leichteste Schnittkante hinzu
- Der MST ist immer zusammenhängend
- **Implementierung:** Markiere alle Knoten außer Startknoten s als unbesucht und die Distanz als unendlich – besuche alle unbesuchten Nachbarknoten und merke dir ihre Distanz – nehme den mit der geringsten, markiere ihn als besucht und füge ihn zum Set A hinzu
- Heap / Priority Queue: Menge von (key- value) – Paaren mit Prioritäten
  - Insert, getMin, deleteMin, decreaseKey
  - Verwalte alle unmarkierten Knoten in einer priority queue; solange Knoten in der queue sind läuft der Algorithmus weiter
  - In value wird die Distanz gespeichert und kann mit decreaseKey() herabgesetzt werden, wenn sie näher an A
- Laufzeit: $O(m \log n)$ mit Heap

5.13 Kruskal's MST-Algorithmus

- ist ein Greedy Algorithmus
- Sortiere Kanten nach Gewicht – iteriere über Kanten – füge hinzu, wenn sie keinen Zyklus ergeben

```
A = ∅
while A ist kein Spannbaum do
    e = {u,v} ist Kante mit kleinstem Gewicht,
    so dass A ∪ {{u,v}} keinen Zyklus enthält
    A = A ∪ {{u,v}}
```

Laufzeit

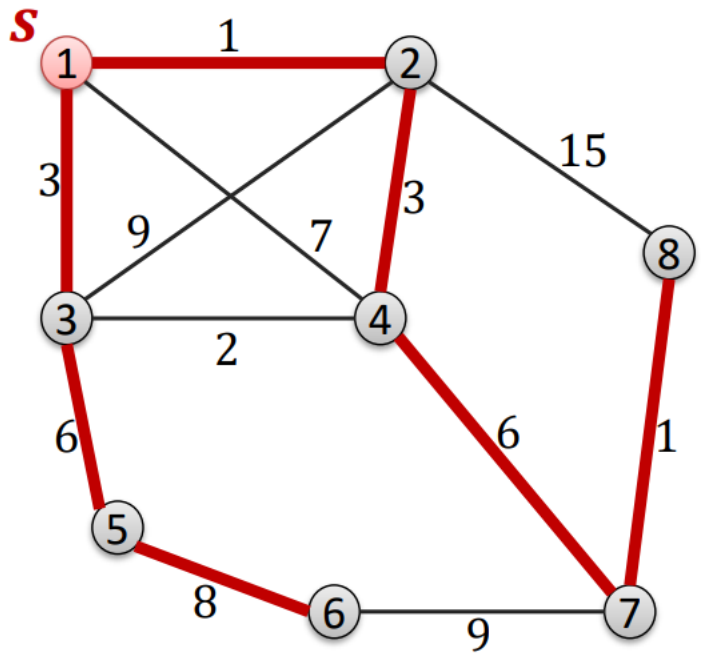
$O(m \log n)$ für das Sortieren, sowie die Gesamtzeit, um die Komponenten zu verwalten...

6 Kürzeste Wege

- gewichteter Graph $G = (V, E, w)$, Startknoten $s \in V$
- Finde kürzeste Pfade / Distanzen von s zu allen Knoten
- **Lemma:** Falls $v_0, v_1, .., v_k$ ein kürzester Pfad von v_0 nach v_k ist, dann gilt für alle $0 \leq i \leq j \leq k$, dass der Teilpfad $v_i, v_{i+1}, ..., v_j$ ein kürzester Pfad von v_i nach v_j ist.

6.1 Shortest-Path Tree

- Mit einem "Shortest-Path Tree" lassen sich die kürzesten Pfade von S aus angeben
- Einen solchen Baum gibt es immer (folgt aus der Optimalität von Teilpfaden)
- Bei ungewichteten Graphen: BFS-Spannbaum



6.2 Dijkstras Algorithmus

- Starte mit s als Teilbaum
- Füge in jedem Schritt den Knoten mit kleinstem Abstand zu s zum Teilbaum hinzu
- Invariante: Algorithmus hat zu jeder Zeit einen von s gewurzelten Teilbaum eines "Shortest Path Tree"

Implementierung

1. Setze zu Beginn T leer und die Distanz zu allen Knoten (außer s) auf unendlich
2. Berechne in jedem Schritt die Distanz zu den Nachbarknoten v, die nicht bereits im Teilbaum sind, über einen Knoten u, der bereits im Teilbaum ist
3. Füge den Knoten mit geringstem Gewicht $d(s, v) = d(s, u) + w(u, v)$ hinzu
4. Die Distanz wird immer in den Knoten gespeichert und in jedem Schritt neu (mit dem neuen Teilbaum) berechne

```
1   H = new priority queue; A = ∅
2   for all u ∈ V/{s} do
3       H.insert(u, ∞); δ(s,u) = ∞; α(u) = NULL
4   H.insert(s, 0)
5   while H is not empty do
6       u = H.deleteMin()
7       for all unmarked out−neighbors v of u do
8           if δ(s,u) + w(u,v) < δ(s,v) then
9               δ(s,v) = δ(s,u) + w(u,v)
10              H.decreaseKey(v, δ(s,v))
11              α(v) = u
12      u.marked = true
13      if u ≠ s then A = A ∪ {(α(u), u)}
```

Analyse

- Laufzeit mit binären Heaps: $O(m \log n)$
- Laufzeit mit Fibonacci Heaps: $O(m + n \log n)$

6.3 Negatives Kantengewicht

Lemma: In einem gerichteten, gewichteten Graphen G hat es genau dann einen kürzesten Pfad von s nach v, falls es keinen negativen Kreis gibt, welcher von s erreichbar ist und von welchem v erreichbar ist.
Einige Algorithmen haben Probleme mit negativen Kanten. Darunter auch Dijkstra.

6.4 Bellman-Ford Algorithmus

- Kürzester Pfad mit n Kanten benötigt n Wiederholungen
- Betrachte alle Kanten (u, v) und versuche $\delta(s, v)$ zu verbessern
- Falls der Graph einen negativen Kreis enthält, gibt der Algorithmus "false" aus

Implementierung

```
1   δ(s,s) := 0; ∀v ∈ V/{s} : δ(s,v) := ∞
2   for i := 1 to n−1 do
3       for all u,v ∈ E do
4           if δ(s,u) + w(u,v) < δ(s,v) then
5               δ(s,v) := δ(s,u) + w(u,v)
6   for all (u,v) ∈ E do
7       if δ(s,u) + w(u,v) < δ(s,v) then
8           return false
9   return true
```

6.5 Routing in Netzwerken

- Knoten merken sich aktuelle Distanz und aktuell besten Nachbarn
- Alle Knoten schauen gleichzeitig, ob es irgendeinen Nachbarn mit Verbesserungen gibt
- Dijkstra bräuchte $O(mn + n^2 \log n)$, geht jedoch nicht für negative Kantengewichte
- Sehr langsam für Bellman-Ford $O(n^4)$

7 Dynamische Programmierung

- Wichtige Algorithmenentwurf-Technik!
- Viele Probleme, welche naiv exponentielle Zeit benötigen, können mit dynamischer Programmierung in polynomieller Zeit gelöst werden
- Entworfen von Richard E. Bellman

7.1 Vorgehensweise

5 Schritte	**Analyse**
1) Teilprobleme definieren	#Teilprobleme zählen
2) Raten (Teil der Lösung)	#Möglichkeiten zählen
4) Rekursion + Memoization	Zeit = Zeit pro Teilproblem *
oder	#Teilprobleme
Bottom-Up DP Tabelle aufbauen	
5) Löse ursprüngliches Problem	Benötigt evtl. zusätzliche Zeit

7.2 Fibonacci Zahlen

```
1  def fib(n):
2      if n < 2:
3          f = n
4      else:
5          f = fib(n-1) + fib(n-2)
6      return f
```

Hier beträgt die Laufzeit $\Omega(2^{n/2})$
Da hier immer wieder gleiche werte berechnet werden, kann man den rekursiven Algorithmus mithilfe von Memoization.

Memoization

```
1  memo = {}
2  def fib(n):
3      if n in memo: return memo[n]
4      if n < 2:
5          f = n
6      else:
7          f = fib(n-1) + fib(n-2)
8      memo[n] = f
9      return f
```

Dadurch, dass schon berechnete Werte gespeichert werden, erspart man sich viel Zeit.
Die Laufzeit beträgt nun $O(n)$

Bottom-up

```
def fib(n):
    fn = {}
    for k in [0,1, 2, ..., n]:
        if k < 2:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```

Bei den Fibonaccizahlen speziell, kann man die Rekursion komplett weglassen. Man benutzt nun ein Array, dass man mit den vorigen Werten einfach auffüllt.

7.3 Kürzeste Wege

Kürzeste Wege können mithilfe von Rekursion berechnet werden

```
dist(v):
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E:
            d = min(d, dist(u) + w(u,v))
    return d
```

Dieser Algorithmus hat aber Probleme mit Zyklen, da hier eine endlose Rekursion entsteht.

Memoization

```
memo = {}
dist(v):
    if v in memo: return memo[v]
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E: (gehe durch alle eingehenden Kanten von v)
        d = min(d, dist(u) + w(u,v))
    memo[v] = d
    return d
```

Durch das einfache Speichern von zuvor berechnetetn Werten lässt sich die Laufzeit auf $O(m)$ reduzieren

Bottom-up

```
D = "array of length n"
for i in [1:n]:
    D[i] = ∞
    if v_i == s:
        D[i] = 0
    else:
        for (v_j, v_i) in E: (eingehende Kanten, top. Sortierung → j < i)
        D[i] = min(D[i], D[j] + w(v_j, v_i))
```

Es wird angenommen, dass die Reihenfolge der Knoten $v_1, v_2, ..., v_n$ topologische sortiert ist

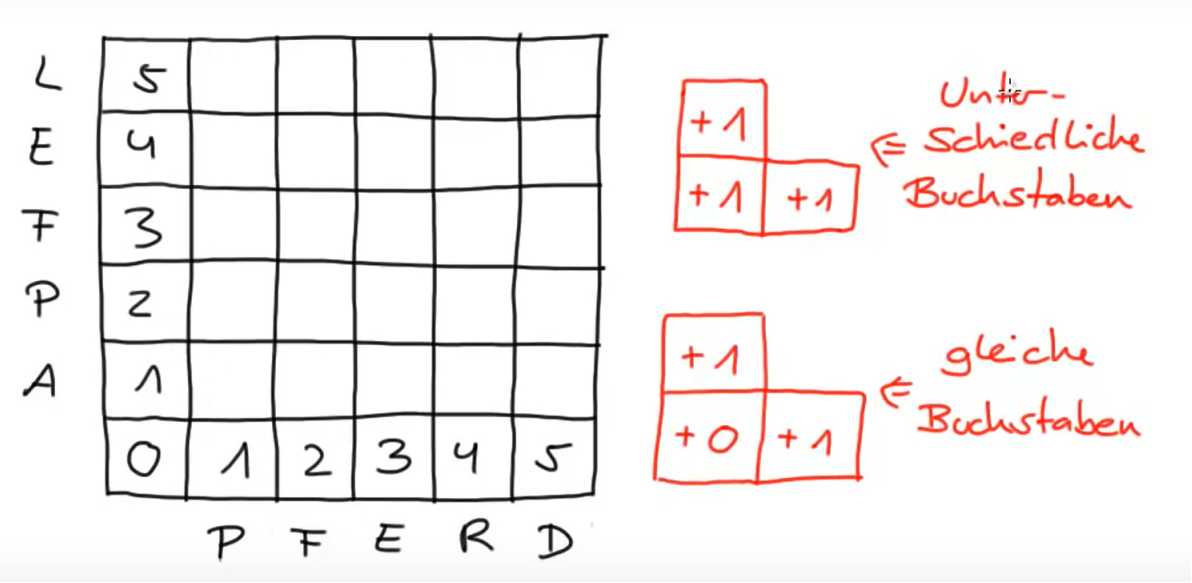
8 Textsuche

8.1 Editierdistanz

Für zwei Zeichenketten A und B, berechne Editierdistanz $D(A, B)$ (# Editierop., um A in B zu überführen) und auch eine minimale Sequenz von Operationen, die A in B überführt.
Mögiche Operationen:

- Ersetzen
- Löschen
- Einfügen

Laufzeit $O(m \cdot n)$



8.2 Textsuche

- Gegeben:
  - Zwei Zeichenketten (Strings)
  - Text T (typischerweise lang)
  - Muster P (engl. pattern, typischerweise kurz)
- Ziel:
  - Finde alle Vorkommen von P in T
- Annahmen:
  - Länge Text T: n
  - Länge Muster P: m
  - $m << n$

Naiver Algorithmus

```
1  TestPosition(s):
2      t = 0
3      while t < m and T[s+t] == P[t] do
4          t = t + 1
5      return (t == m)
6
7  String-Matching:
```

```
8           for s from 0 to n - m do
9               if TestPosition(s) then
10                  report found match at position s
```

Laufzeit zwischen $O(n \cdot n)$ und $O(n)$

8.3 Rabin-Karp Algorithmus

- Wir schieben wieder ein Fenster der Grösse m über den Text und schauen an jeder Stelle, ob das Muster passt
- Um den Vergleich zu beschläunigen kann man Hashing oder den Modulo benutzten
- Wenn die Werte übereinstimmen, werden die Werte mit dem naiven Algorithmus nochmal vergluchen
- Der wählt als $MambesteneienZweierpotenz$ und aös b einen möglichts großen werte
- Dies verringert die warscheinlichkeit, dass zwei unterschidliche werde dieselbe Zahl haben

```
1       h = b^(m-1) mod M
2       p = 0; t = 0;
3       for i = 0 to m - 1 do
4           p = (p · b + P[i]) mod m
5           t = (t · b + T[i]) mod M
6       for s = 0 to n - m do
7           if p == t then
8               TestPosition(s)
9               t = ((t - T[s] · h) · b + T[s + m]) mod M
```

Die Laufzeit liegt von $O(n \cdot m)$ und $O(n + k \cdot m)$

8.4 Knuth-Morris-Pratt Algorithmus

- Merken, wo es ein Mismatch gab
- Array S gibt an, an welcher Stelle weiter gesucht werden soll

Vorberechnung

```
1       h = S[i-1]
2       while h ≥ 0 do
3           if P[i-1] == P[h] then
4               S[i] = h + 1; h = -2
5           else
6               h = S[h]
7       if h == -1 then S[i] = 0
```

Implementierung

```
1       t = 0; p = 0     // t: Position in Text, p: Position im Pattern
2       while t < n do
3           if T[t] == P[p] then  // characters match
4               if p == m - 1 then // pattern found
5                   pattern found at position t - m + 1
6                   p = S[m] ; t = t + 1
7               else
8                   p = p + 1; t = t + 1
9           else          // characters does not match
10              if p == 0 then       // mismatch at first character
11                  t = t + 1
12              else
13                  p = S[p]
```

Analyse

Laufzeit: $O(m+n) = O(n)$