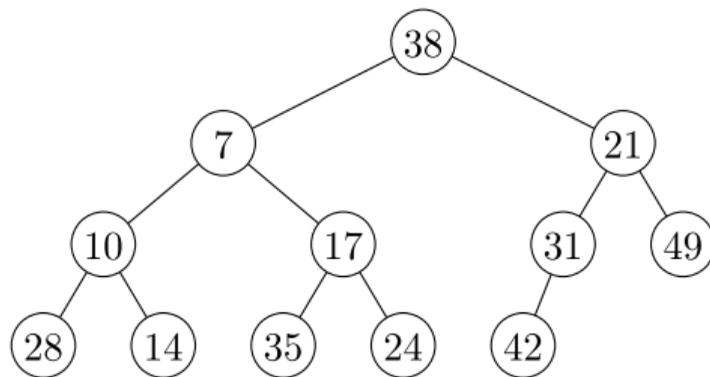


extract-min



Hinuntersinken: **Zwei** Vergleiche pro Schritt.

Alternative:

- 1 **Hinuntersinken** bis zum Blatt.
- 2 Dann von dort **aufsteigen**.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

- 1 Konstruiere einen Max-Heap (größer = oben)
- 2 Entferne wiederholt das größte Element
- 3 Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und n in $O(\log n)$ Zeit

Heapsort ist ein **in-place**-Verfahren.

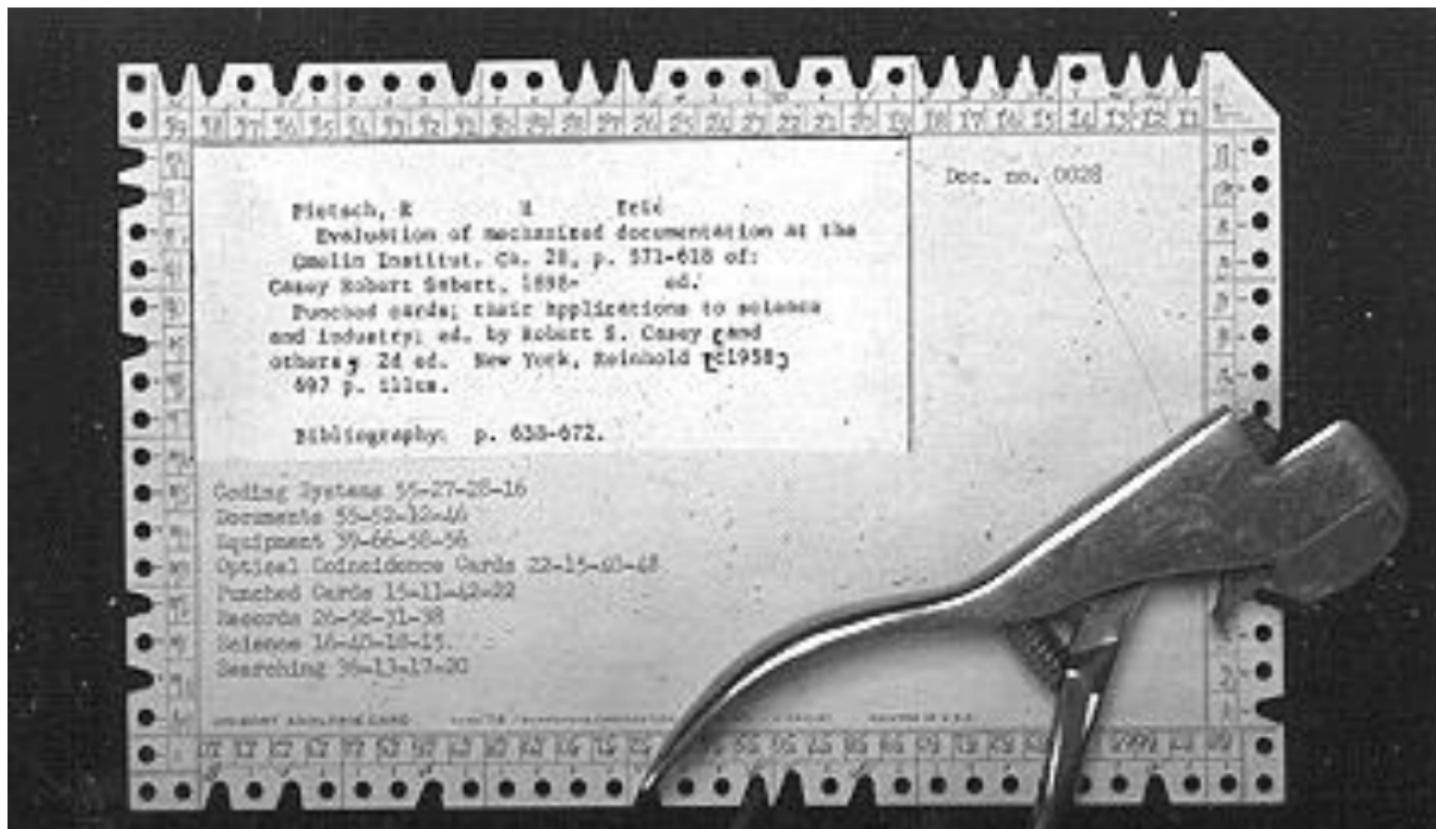
Heapsort – Beispiel



Heapsort – Schnellere Variante



Radixsort



Gegeben seien Binärzahlen einer gewissen Länge:

11101010000010001000	01000000110011010001
10110001100111111001	000000001000000011
01111100111110100100	01111100111110100100
00001011110100010001	00001011110100010001
01011011000110000000	01011011000110000000
11010111000100000110	01101011111000001001
00010111011101000011	00010111011101000011
01101011111000001001	11010111000100000110
10001101100000001100	10001101100000001100
11100000010000001010	11100000010000001010
10101010000011110000	10101010000011110000
10100100001011001010	10100100001011001010
0000000010000000011	10110001100111111001
01000000110011010001	11101010000010001000

Sortiere nach dem ersten Bit.

Bitstrings in Matrix $A[1 \dots n, 1 \dots w]$.

Vor dem Bitarray stehe $00 \dots 0$ und danach $11 \dots 1$.

Algorithmus

procedure radix_exchange_sort(i, a, b) :

if $i > w$ **then return** i ;

$s := a$; $t := b$;

while $s < t$ **do**

while $A[s, i] = 0$ **do** $s := s + 1$;

while $A[t, i] = 1$ **do** $t := t - 1$;

 vertausche $A[s]$ und $A[t]$

od;

vertausche $A[s]$ und $A[t]$;

radix_exchange_sort($i + 1, a, t$);

radix_exchange_sort($i + 1, s, b$)

```

11101010000010001000
10110001100111111001
01111100111110100100
00001011110100010001
01011011000110000000
11010111000100000110
00010111011101000011
01101011111000001001
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
00000000010000000011
01000000110011010001

```

```

11101010000010001000
01111100111110100100
01011011000110000000
11010111000100000110
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
10110001100111111001
00001011110100010001
00010111011101000011
01101011111000001001
00000000010000000011
01000000110011010001

```

Sortiere nach dem letzten Bit.

Dann nach dem vorletzten usw. (→ stabil!)

Straight-Radix-Sort

Algorithmus

```
procedure straight_radix_sort(i) :  
  pos0 := 1; pos1 := n + 1;  
  for k = 1, ..., n do pos1 := pos1 - A[k, i] od;  
  for k = 1, ..., n do  
    if A[k, i] = 0 then B[pos0] := A[k]; pos0 := pos0 + 1  
      else B[pos1] := A[k]; pos1 := pos1 + 1 fi  
  od
```

Sortiere stabil nach dem i -ten Bit.

Ergebnis ist in n.

Straight-Radix-Sort

Vollständiger Algorithmus:

Algorithmus

```
procedure straight_radix_sort :
```

```
  for  $i = w, \dots, 1$  do
```

```
     $pos0 := 1; pos1 := n + 1;$ 
```

```
    for  $k = 1, \dots, n$  do  $pos1 := pos1 - A[k, i]$  od;
```

```
    for  $k = 1, \dots, n$  do
```

```
      if  $A[k, i] = 0$  then  $B[pos0] := A[k]; pos0 := pos0 + 1$ 
```

```
        else  $B[pos1] := A[k]; pos1 := pos1 + 1$  fi
```

```
    od;
```

```
     $A := B;$ 
```

```
  od;
```

Übersicht

- 2 Suchen und Sortieren
 - Einfache Suche
 - Binäre Suchbäume
 - Hashing
 - Skip-Lists
 - Mengen
 - Sortieren
 - **Order-Statistics**

Order-Statistics

Eingabe:

Eine Menge von n Schlüsseln aus einer geordneten Menge

Eine Zahl k , $1 \leq k \leq n$

Ausgabe:

Der k -te Schlüssel (nach Größe)

Spezialfall:

Median, der Schlüssel in der Mitte.

Einfachste Lösung:

- 1 Sortieren
- 2 Den Schlüssel an Position k zurückgeben

Quickselect

Wie Quicksort, aber nur die **richtige** Seite rekursiv behandeln:

Algorithmus

```
procedure quickselect(k, L, R) :  
  if  $R \leq L$  then return a[k] fi;  
  p := a[L]; l := L; r := R + 1;  
  do  
    do l := l + 1 while a[l] < p;  
    do r := r - 1 while p < a[r];  
    vertausche a[l] und a[r]  
  while l < r;  
  temp := a[r]; a[L] := a[l]; a[l] := temp; a[r] := p;  
  if k = r then return p  
  else if k < r then return quickselect(k, L, r)  
  else return quickselect(k, r, R) fi
```

Quickselect – Analyse

Bei Quicksort hatten wir diese Rekursionsgleichung für die Anzahl der Vergleiche:

$$C_n = n + 1 + \frac{1}{n} \sum_{i=1}^n (C_{i-1} + C_{n-i}).$$

Für Quickselect gilt:

- Mit W'keit $1/n$ ist das Pivotelement der gesuchte Schlüssel
- Mit W'keit $(k-1)/n$ ist der gesuchte Schlüssel **links**
- Mit W'keit $(n-k)/n$ ist der gesuchte Schlüssel **rechts**

Quickselect – Analyse

Für $n > 1$ haben wir:

$$\begin{aligned} C_n &= n + 1 + \frac{1}{n} \sum_{i=1}^{k-1} C_{n-i} + \frac{1}{n} \sum_{i=k+1}^n C_{i-1} \\ &\leq n + 1 + \frac{2}{n} \sum_{i=\lceil n/2 \rceil}^{n-1} C_i \end{aligned}$$

Außerdem ist $C_0 = C_1 = 0$.

Zeige mit Induktion:

$$C_n \leq 4n$$

(Einfach \rightarrow Übungsaufgabe)

Quickselect

Theorem

*Quickselect findet den Schlüssel mit Rang k in einer n -elementigen Menge in $O(n)$ Schritten – **im Erwartungswert**.*

Ein deterministischer Algorithmus

- 1 Falls $n < 30$: Sortiere und finde so das Ergebnis.
- 2 Bilde $m = \lfloor n/5 \rfloor$ Gruppen mit je fünf Schlüsseln. Bis zu vier Schlüssel bleiben übrig.
- 3 Berechne den Median jeder Gruppe.
- 4 Berechne rekursiv den Median p dieser $\lfloor n/5 \rfloor$ Mediane.
- 5 Verwende p als Pivotelement und führe damit Quickselect aus.

Welchen Rang r hat p ?

p ist der Median von $m = \lfloor n/5 \rfloor$ vielen kleinen Medianen

→ mindestens $m/2 - 1$ kleine Mediane sind kleiner als p

Jeder kleine Median hat zwei Schlüssel die kleiner sind

→ mindestens $3m/2 - 1$ kleinere Schlüssel als p

Ebenso: Mindestens $3m/2 - 1$ größere Schlüssel als p

Das sind jeweils $3\lfloor n/5 \rfloor / 2 - 1 \geq 3n/10 - 3/2 \geq n/4$

Deterministisches Selektieren – Analyse

Die Anzahl der Vergleiche ist jetzt

$$C_n \leq n + 1 + C_{\lfloor n/5 \rfloor} + C_{\lfloor 3n/4 \rfloor}$$

falls $n > 30$ und $O(1)$ falls $n \leq 30$:

- $C_{\lfloor n/5 \rfloor}$ für das rekursive Finden der Mediane
- $C_{\lfloor 3n/4 \rfloor}$ für die nächste Suche

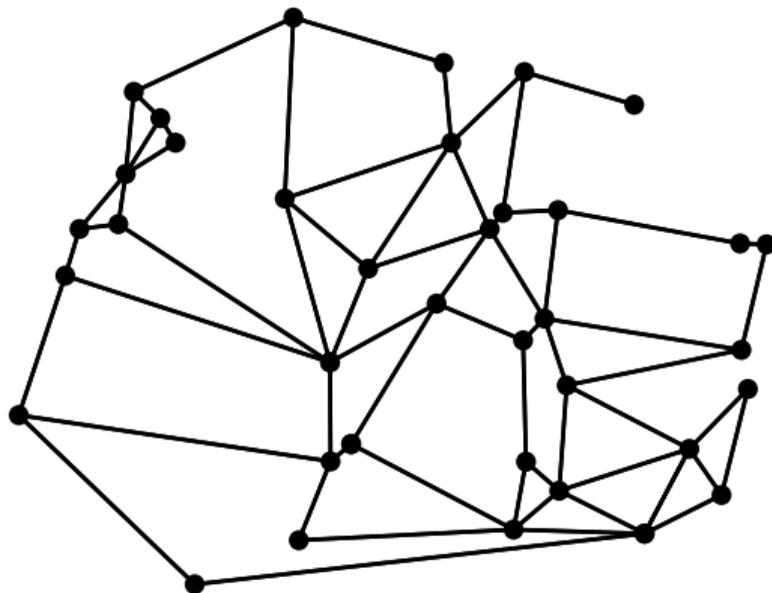
Es folgt $C_n = O(n)$, da $1/5 + 3/4 < 1$.

Wir können also den Schlüssel mit Rang k in linearer Zeit finden.

Übersicht

- 1 Einführung
- 2 Suchen und Sortieren
- 3 Graphalgorithmen**
- 4 Algorithmische Geometrie
- 5 Textalgorithmen
- 6 Paradigmen

Graphen



Graphen

Definition

Ein **ungerichteter Graph** ist ein Paar (V, E) , wobei V die Menge der **Knoten** und $E \subseteq \binom{V}{2}$ die Menge der **Kanten** ist.

Definition

Ein **gerichteter Graph** ist ein Paar (V, E) , wobei V die Menge der **Knoten** und $E \subseteq V \times V$ die Menge der **Kanten** ist.

Oft betrachten wir Graphen mit Knoten- oder Kantengewichten.

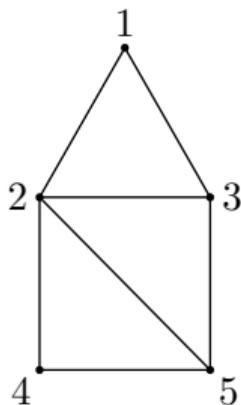
Dann gibt es zusätzlich Funktionen $V \rightarrow \mathbf{R}$ oder $E \rightarrow \mathbf{R}$.

Übersicht

- 3 Graphalgorithmen
 - Darstellung von Graphen
 - Tiefensuche
 - Starke Komponenten
 - Topologisches Sortieren
 - Kürzeste Pfade
 - Netzwerkalgorithmen
 - Minimale Spann bäume

Darstellung von Graphen

Adjazenzmatrix

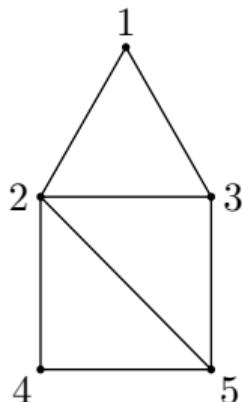


$$\begin{pmatrix} \cdot & 1 & 1 & 0 & 0 \\ \cdot & \cdot & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 0 & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Speicherbedarf: $\Theta(|V|^2)$

Für gerichtete Graphen wird die ganze Matrix verwendet.

Darstellung von Graphen



Adjazenzliste

1		2, 3
2		1, 3, 4, 5
3		1, 2, 5
4		2, 5
5		2, 3, 4

Speicherbedarf: $\Theta(|V| + |E|)$.

In $O(n^2)$ Schritten kann zwischen beiden Darstellungen konvertiert werden.

Darstellung von Graphen

Java

```
public class SimpleGraph<V> implements Graph<V> {  
    protected Set<V> nodes;  
    protected Map<V, List<V>> edges;  
    protected boolean directed = false;  
    protected Map<String, Map<V, Object>> nodeAttributes;  
    protected Map<String, Map<Edge<V>, Object>> edgeAttributes;  
    public SimpleGraph() {  
        nodes = new HashSet<V>();  
        edges = new HashMap<V, List<V>>();  
        nodeAttributes = new HashMap<>();  
        edgeAttributes = new HashMap<>();  
    }  
}
```

Java

```
public class Edge<V> {  
    public V s, t;  
    private boolean directed;  
    protected Edge(V s, V t, boolean directed) {  
        this.s = s;  
        this.t = t;  
        this.directed = directed;  
    }  
}
```

Java

```
public void addNode(V u) {  
    nodes.add(u);  
    edges.put(u, new LinkedList<V>());  
}
```

Java

```
public void addEdge(V s, V t) {  
    List<V> adjlist = edges.get(s);  
    adjlist.add(t);  
    if(!directed) {  
        adjlist = edges.get(t);  
        adjlist.add(s);  
    }  
}
```

Java

```
public void delEdge(V s, V t) {  
    List<V> adjlist = edges.get(s);  
    adjlist.remove(t);  
    if(!directed) {  
        adjlist = edges.get(t);  
        adjlist.remove(s);  
    }  
}
```