University of Basel

Randomisierte Algorithmen und Datenstrukturen

Marcel Lüthi

Randomisierter Algorithmus

- Mehrere Ausführungen des *gleichen* randomisierten Algorithmus für die gleiche Eingabe können *verschiedene Ergebnisse* liefern.

- Laufzeit und Korrektheit nur mit gewisser Wahrscheinlichkeit bestimmbar.

- Sehr oft einfacher zu implementieren als deterministischer Algorithmus.
  - Manchmal effizienter.

- Erlaubt oft elegante (probabilistische) Lösungen zu schwierigen Problemen.

Kategorien randomisierter Algorithmen

Numerische Algorithmen

- Lösung ist nur approximativ korrekt
- Typischerweise kann man Konfidenzinterval angeben:
  – Mit Wahrscheinlichkeit 0.999 ist die korrekte Lösung 59 ± 3.

Monte Carlo Algorithmen

- Lösung ist mit hoher Wahrscheinlichkeit korrekt
- Manchmal ist Lösung aber inkorrekt.

Las Vegas Algorithmen

- Lösung ist immer korrekt
- Aber, kann auch mal keine Lösung liefern.

Beispiel

Wann hat Christoph Kolumbus Amerika entdeckt?

Numerischer Algorithmus:

- Zwischen 1490 und 1500
- Zwischen 1485 und 1495
- Zwischen 1491 und 1501
- Zwischen 1480 und 1490
- Zwischen 1489 und 1499

Monte Carlo Algorithmus:

- 1492, 1492, 1492, 1491, 1492, 1492, 1291, 1492, 1492, 1492

Las Vegas Algorithmus:

- 1492, 1492, 1492, Sorry!, 1492, 1492, 1492, 1492, Sorry!, 1492

Randomisierte Datenstrukturen

Bei gleichem Input kann die «Form» der Datenstruktur jedes mal anders aussehen.
Komplexitätsgarantien gelten nur probabilistisch

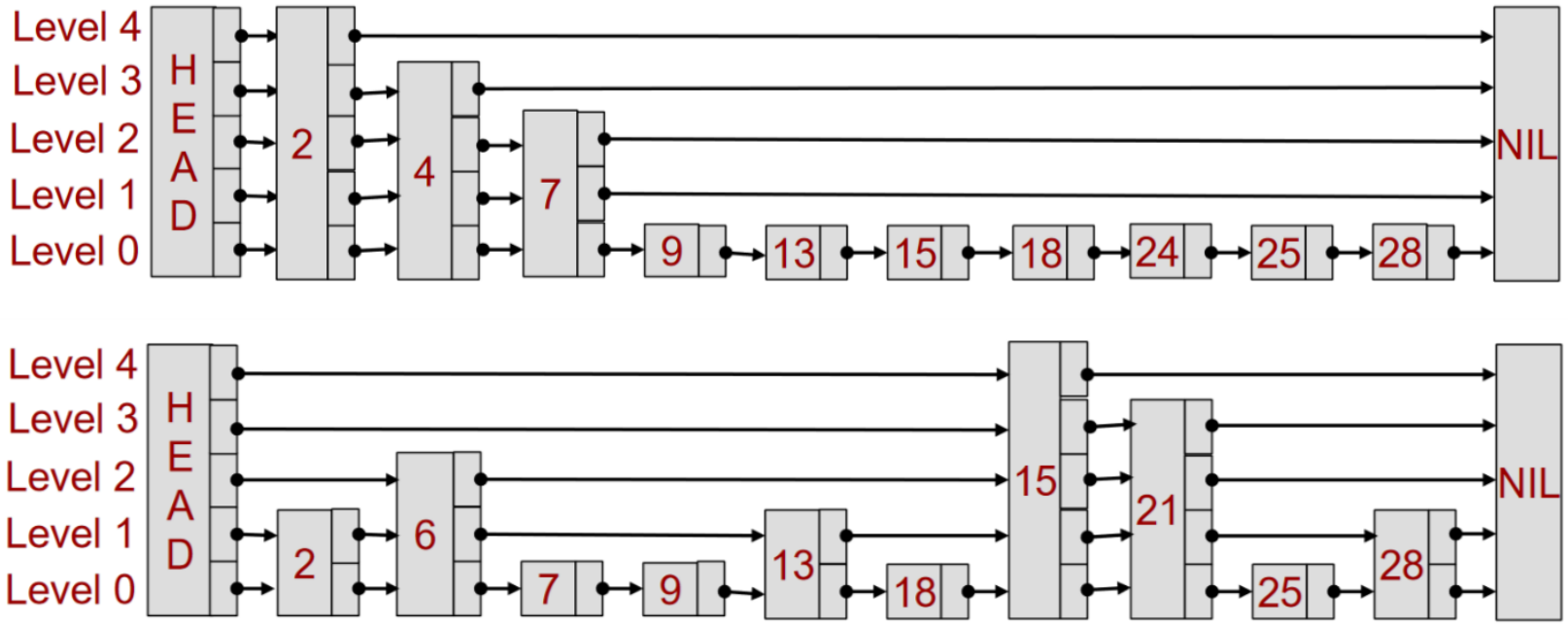


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Numerische Algorithmen

π schätzen

Gegeben:

Kreis von Radius 0.5

Quadrat: Seitenlänge 1

Fläche Kreis: $\pi r^2 = \frac{\pi}{4}$

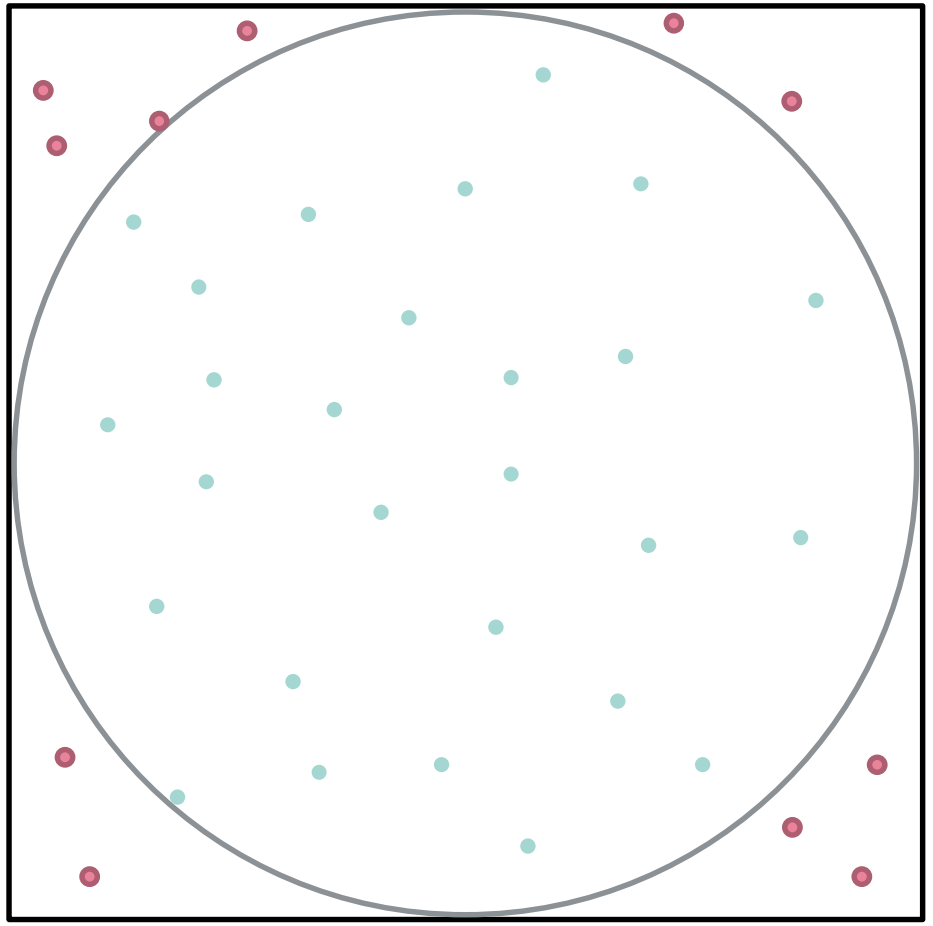
Beobachtung:

$$\frac{\text{Fläche Quadrat}}{\text{Fläche Kreis}} = \frac{4}{\pi}$$

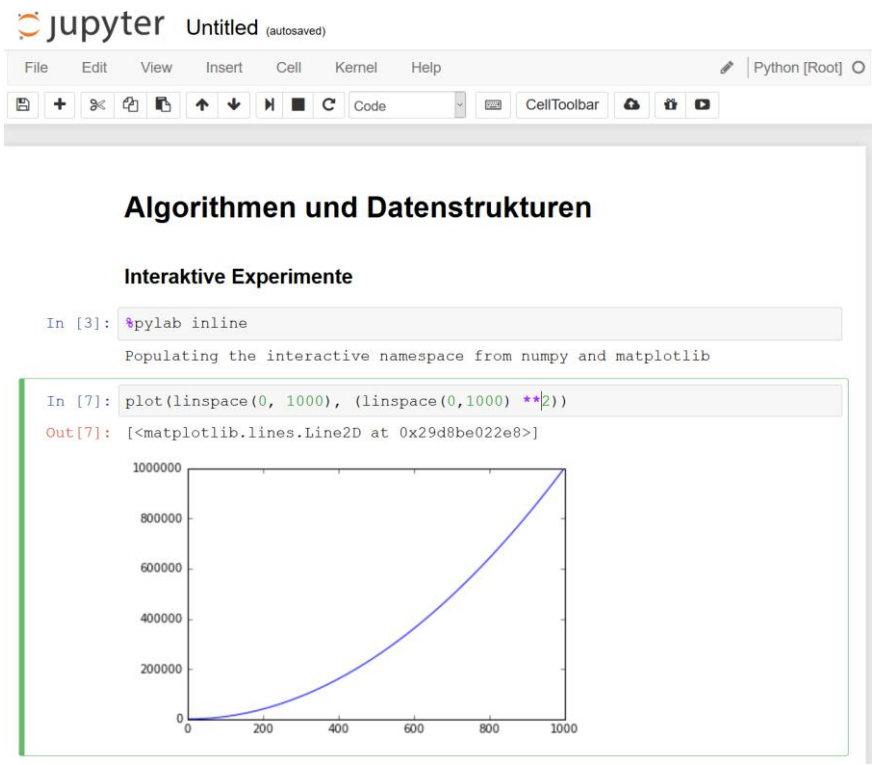
Algorithmus:

- Ziehe zufällige (gleichverteilte) Punkte im Quadrat.
- Berechne:

$$\pi \approx \frac{\#\ samples\ in\ Kreis}{\#\ samples} \cdot 4$$



Implementation



Randomized.ipynb

Monte Carlo Algorithmen

Primzahlen testen

Problem:

- Ist eine gegebene (ungerade) Zahl eine Primzahl?

Erster Ansatz:

Kleiner Satz von Fermat

Sei n eine Primzahl. Dann gilt

$$a^{n-1} mod\ n = 1$$

für jede Ganzzahl $a \in \mathbb{Z}$ mit $1 \leq a \leq n-1$

Schlussfolgerung:

- Falls n und a Ganzzahlen sind mit $1 \leq a \leq n-1$ und $a^{n-1} mod\ n \neq 1$ dann ist n keine Primzahl.

Primzahlen testen

```
def fermat(n):
    a = random.randint(1, n-1)
    return a**(n-1) % n  == 1
```

Test ist nicht zuverlässig!

- $(n-1)^{n-1}\ mod\ n = 1$ für alle ungeraden $n \geq 3$

Beispiel:

$14^{14}\ mod\ 15 = 1$ aber 15 ist keine Primzahl

$(n-1)^{n-1}\ mod\ n = 1$ ist ein falscher Zeuge für n.

Primzahlen testen

Idee:

Angenomme Fehlerwahrscheinlichkeit ist nach oben beschränkt:

$p_n \leq p$ für alle n

Wiederhole Test n mal
– False ⇒ n ist keine Primzahl
– True ⇒ Wahrscheinlichkeit, dass n Primzahl ist: p^k klein, dass es Primzahl ist.

```
def repeatFermat(n, k):
    for i in range(0, k):
        if not fermat(n):
            return False
    return True
```

Primzahlen testen

Definition*:*

Sei $0 < p < 1$*. Ein Monte-Carlo Algorithmus ist* ***p-korrekt****, falls die Wahrscheinlichkeit ein korrektes Resultat zu liefern für jede Eingabe mindestens* p *ist.*

Fehlerwahrscheinlichkeit in Zahlen bis 1000 < 3 %
Aber:
– 561 hat 318 falsche Zeugen.
– femat(6516930556936 81) hat Wahrscheinlichkeit > 99.965 obwohl es zusammengesetzte Zahl ist.

Methode *fermat* ist nicht p-korrekt ist.

Milner-Rabin Methode für Primzahlentest

Sei n > 4, n ungrade und s, $t, so\ dass\ n - 1 = 2^s t, t$ ungerade.

Definiere Menge $B(n)$ wie folgt:

- $a \in B(n)$ genau dann wenn $2 \leq a \leq n - 1$ und
  - $a^t mod\ n = 1$
    oder
  - $\exists i, 0 < i < s$, so dass $a^{2^i t} mod\ n = n - 1$

Satz:

Sei $n > 4$ eine beliebige ungerade Zahl.

- *Falls n Primzahl ist, dann ist $B(n) = \{a | 2 \leq a \leq n - 2\}$*
- *Falls n zusammengesetzte Zahl ist, ist $|B(n)| \leq (n - 9)/4$*

Milner-Rabin Methode für Primzahlentest

```
def bTest(a, n):
    s = 1
    t = n - 1
    while not t % 2 == 1:
        s = s + 1
        t = t / 2
    x = a**t % n
    if x == 1 or x == n - 1:
        return True
    for i in range(0, s):
        x = x**2 % n
        if x == n - 1:
            return True
    return False
```

- Gibt immer true zurück falls n Primzahl > 4 und $2 \leq a \leq n-2$
- Falsch mit Wahrscheinlichkeit ¾ wenn n zusammengesetzte ungerade Zahl > 4 und a zufällig zwischen 2 und $n-2$ gewählt wird.

Milner-Rabin Methode für Primzahlentest

Folgender Algorithmus ist $\frac{3}{4}$ –Korrekt:

```
def millerRabin(n):
    a = randint(2, n-1)
    return bTest(a, n)
```

Folgender Algorithmus kann beliebige Genauigkeit erreichen:

```
def repeatMillerRabin(n, k):
    for i in range(0, k):
        if millerRabin(n):
            return False
    return True
```

Las-Vegas Algorithmen

Las-Vegas Algorithmen

Zufall wird benutzt um schneller zu korrekter Lösung zu kommen.

Wichtiger Unterschied zu Monte-Carlo Algorithmen:
– Antwort ist immer korrekt.

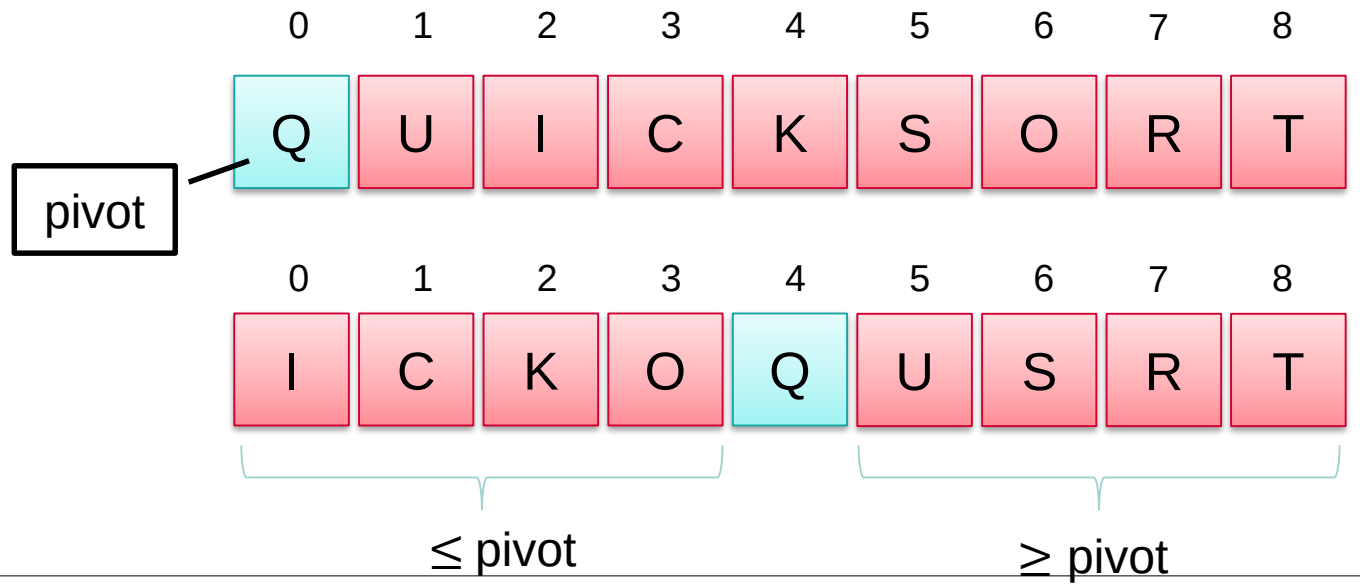
Zwei Kategorien:
1. Falls schlechte Wahl getroffen wird, dauert es länger Lösung zu berechnen
2. Falls schlechte Wahl getroffen wird, gibt es keine Lösung.

Laufzeit ist (im Schnitt) nicht besser als bei deterministischem Algorithmus.

Quicksort

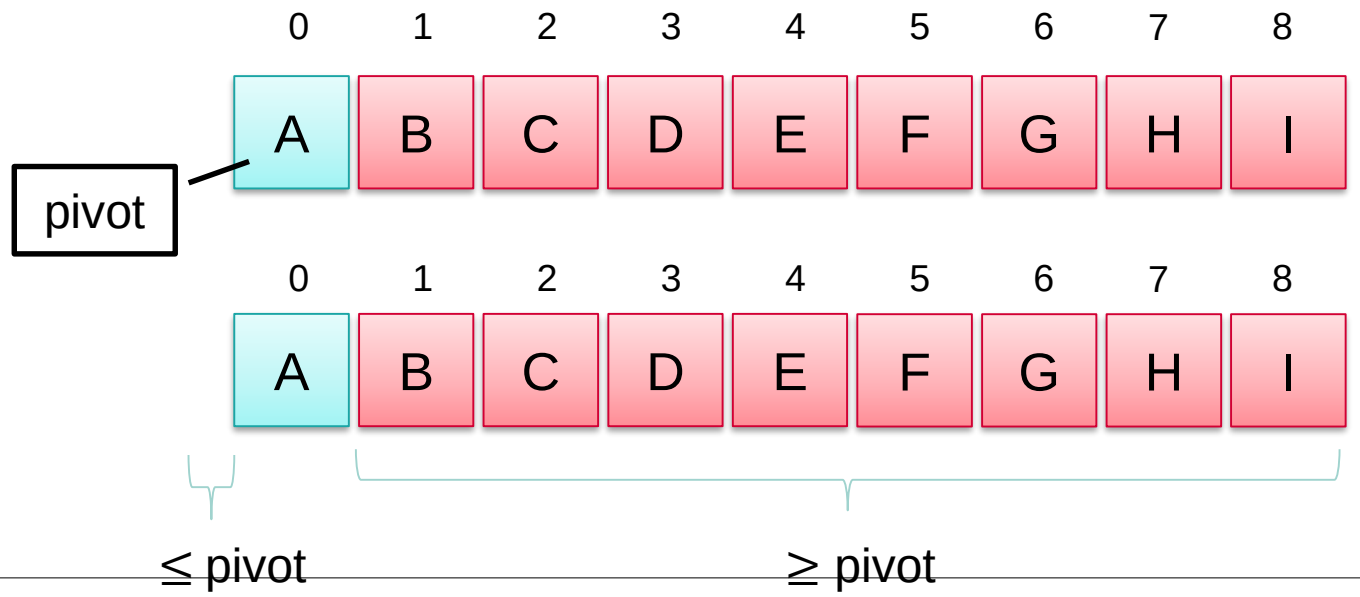
Strategie:

– Wähle erstes Element als Pivot
– Partitioniere
Komplexität (Average case): $O(n \log n)$



Quicksort

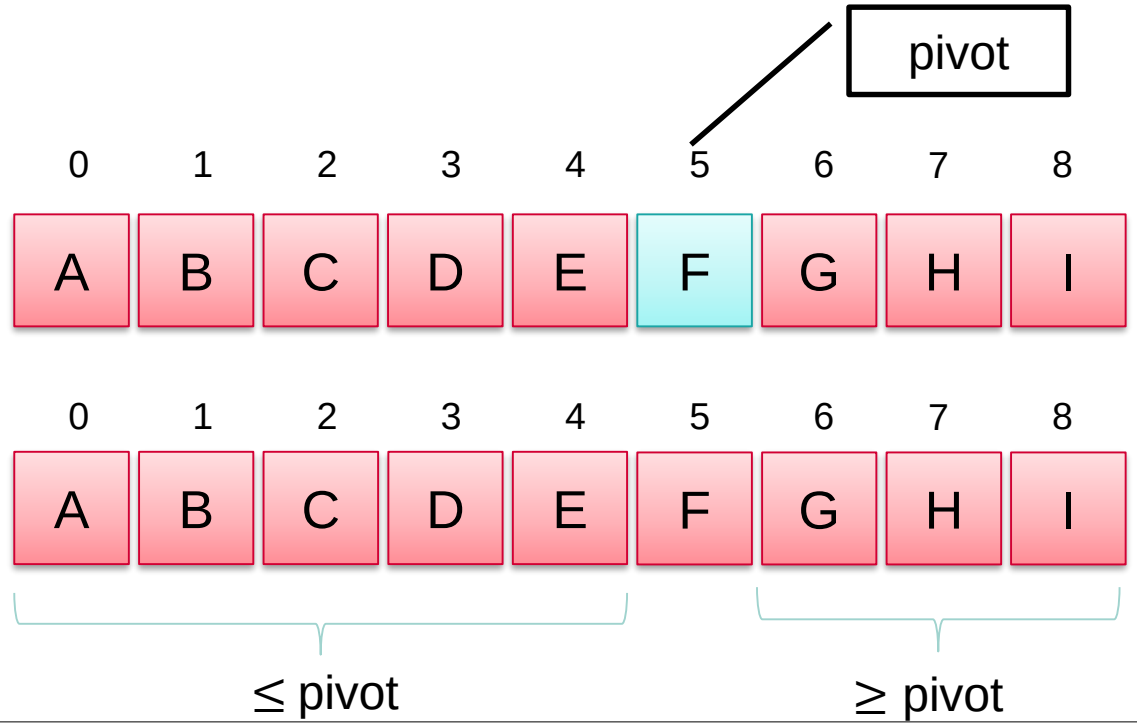
Problem: Laufzeit hängt von Sequenz ab
Komplexität (Worst case): $O(n^2)$



Randomisierter Quicksort

Lösung: Wähle Pivot Element zufällig
Laufzeit: Worst case **Erwartungswert** $O(n \log n)$

Nicht über alle möglichen Eingaben gemittelt, sondern gilt für jede Instanz



Randomisierter Quicksort

Laufzeit nicht mehr deterministisch durch Sequenz der Eingabedaten bestimmt

- $O(n^2)$ noch immer möglich aber extrem unwahrscheinlich
- Wahrscheinlichkeit für worst-case: $\frac{2^{n-1}}{n!}$

Paradoxe Situation:

- *Einführen von Zufallselement gibt uns mehr Kontrolle über Laufzeit.*

Randomisierte Datenstrukturen

Skip-List

- Sortierte Verkettete Liste mit Abkürzungen
- Erlaubt schnelles Suchen / Einfügen

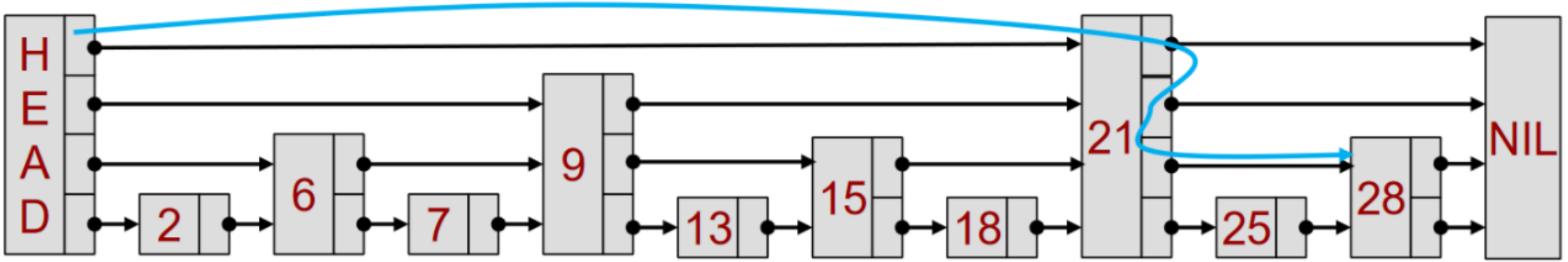


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Perfekte Skip-List

- Beginne mit normaler verketteter Liste (Ebene 0)
- Jeder 2 Knoten wird eine Ebene erhöht und verkettet (Ebene 1)
- Jeder 2 Knote in Ebene 1 wird wieder erhöht und verkettet (Ebenen 2)
- Insgesamt $O(\log_2(n))$ Ebenen

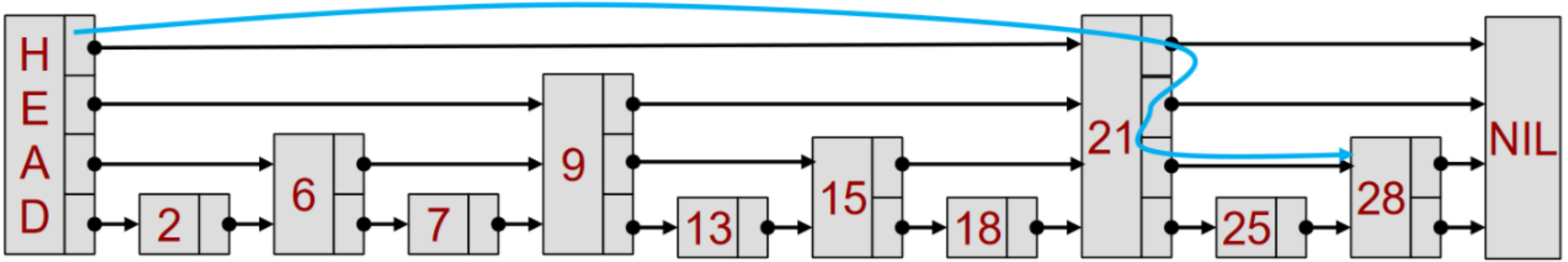


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Suchzeit in der perfekten Skip-List

Satz:
Jeder Knoten kann in $O(\log(n))$ Operationen gefunden werden.

Beweis
– Auf jeder Ebene werden maximal 2 Knoten durchlaufen
– Es gibt $O(\log(n))$ Ebenen
– Also maximal $O\big(2(\log(n))\big)$ Ebenen= $O(\log(n))$

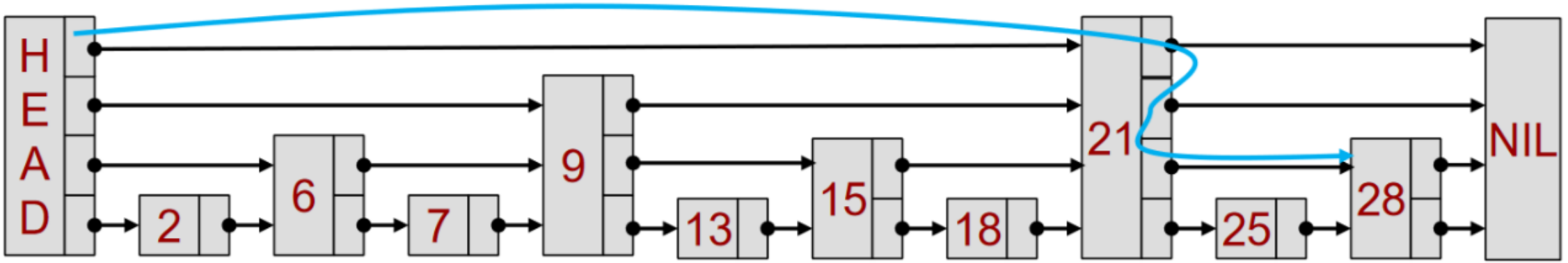


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Das Problem mit der perfekten Skip-Liste

- Was passiert beim Einfügen / Löschen?
- Wieviele Knoten müssten neu verlinkt werden?
  - Worst case: Alle n-1 verbliebenen Knoten

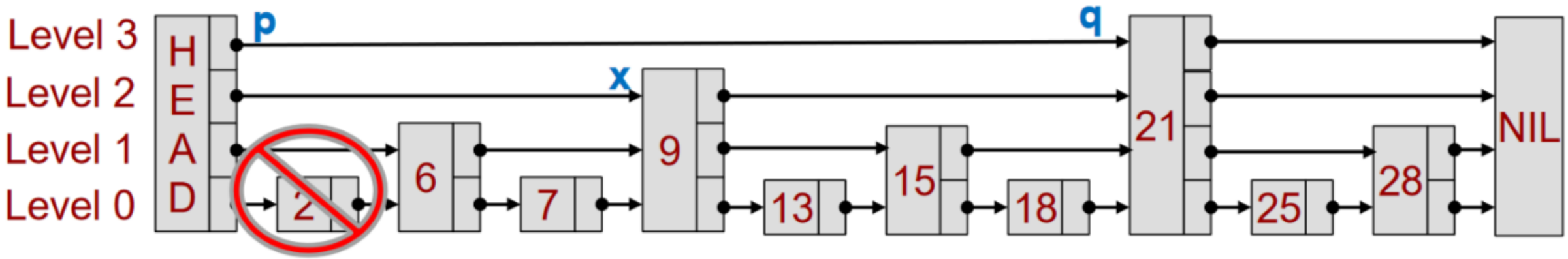


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Münzen werfen

Spiel: Für jedes mal wenn eine Münze Kopf zeigt, erhält man 1 Franken und kann nochmals spielen.

Wie wahrscheinlich ist es
– 1 Franken
– 2 Franken
– 4 Franken
zu gewinnen?

This Photo by Unknown Author is licensed under CC BY

Randomisierte Skip Listen

Für jeden eingefügten Node wird Ebene per Münzwurf entschieden
– Ebene 1, mit Warscheinlichkeit ½
– Ebene 2, mit Warscheinlichkeit ¼
– Ebene 3, mit Warscheinlichkeit $\frac{1}{8}$
Im Erwartungswert immer dieselbe Anzahl Knoten auf jeder Ebene
– Erwartete Komplexität für Einfügen / Löschen und Suche: $O(\log_2(n))$

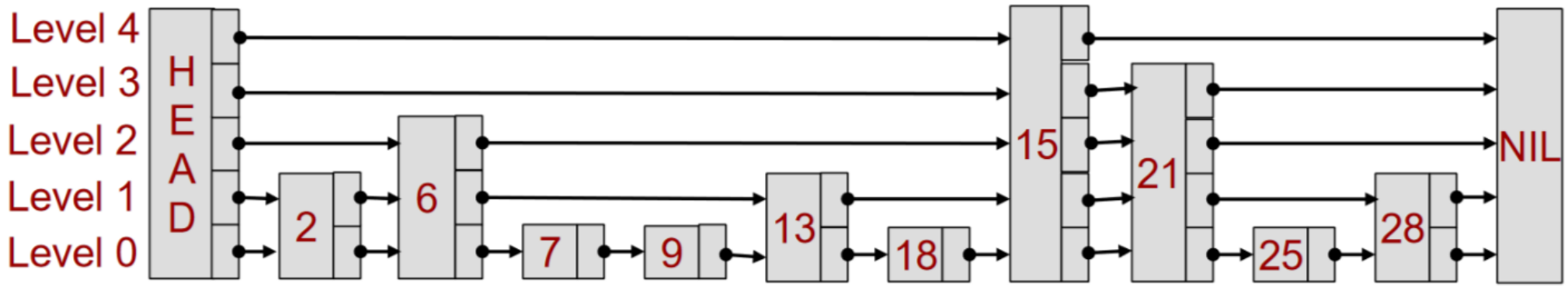


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Worst case

Worst case Komplexität: $O(n)$
Aber: Sehr unwahrscheinlich

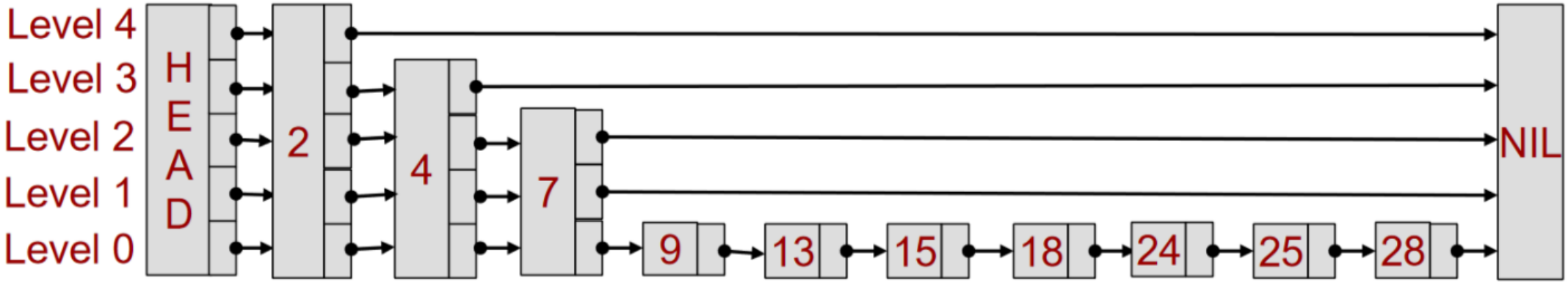


Abbildung: Mark Redekopp, http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

Zusammenfassung

Einbeziehen von Zufallselementen führt zu interessanter Klasse von Algorithmen und Datenstrukturen.
– Oft einfach und elegant
– Manchmal schneller

Schwierige Probleme werden oft einfacher, wenn wir Fehler tolerieren.
Wichtig: Wir akzeptieren Fehler, müssen diese aber gut managen!
– Dann können wir strikte (probabilistische) Garantien geben.
– In praktischer Anwendung manchmal sogar besser als deterministische Lösung.