

Informatik II
Skip-Lists & selbstorganisierende Listen
 (als Bsp. für randomisierte Algos und selbstorganisierende Datenstrukturen)

G. Zachmann
 Clausthal University, Germany
zach@in.tu-clausthal.de

Randomisierte Algorithmen

- Werfen im Verlauf ein- oder mehrmals eine Münze, um Entscheidungen zu treffen
- Enthalten Statements der Form


```
if random() >= 0.5:
    ...
else:
    ...
```
- Folge:
 - Laufzeit hängt i.A. von diesen Münzwürfen ab; kann **verschieden** sein, bei gleicher Eingabe
 - Worst-case Laufzeit ist nicht mehr an bestimmte Eingabe gekoppelt, sondern an bestimmte **Folgen von (zufälligen) Entscheidungen**
- Datenstrukturen, die von randomisierten Algorithmen aufgebaut werden, und je nach Zufall leicht verschiedene konkrete Struktur haben können, heißen auch **randomisierte Datenstrukturen**

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 2

Motivation

- Ausgangspunkt: Suche in geordneten Listen
 - Elemente sind nach einer Ordnungsrelation geordnet
 - binäre Suche nicht möglich, da nur sequentieller Zugriff
- Idee: Beschleunige Suche durch zusätzliche Zeiger

- Knoten haben jetzt auch eine **Höhe** h , $h \geq 0$
- Jeder Knoten hat nicht mehr genau 1 Nachfolger und 1 Zeiger, sondern eine **Liste von Zeigern** (nicht Arrays gleicher Größe!)

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 3

Perfekte Skip-Listen

- Definition: sortierte, verkettete gespeicherte Liste mit Kopfelement (Schlüssel $-\infty$) und Endelement (Schlüssel $+\infty$) und:
 - Jeder 2^h -te Knoten hat Zeiger auf den nächsten Knoten mit Höhe ≥ 0
 - Jeder 2^1 -te Knoten hat Zeiger auf den 2^1 entfernten Knoten mit Höhe ≥ 1
 - Jeder 2^2 -te Knoten hat Zeiger auf den 2^2 entfernten Knoten mit Höhe ≥ 2
 - ...
 - Jeder 2^k -te Knoten hat Zeiger auf den 2^k entfernten Knoten mit Höhe $\geq k$, $k = \lceil \log n \rceil - 1$ ($n =$ Anzahl Knoten $+ 2$ für Head und Tail)
- Und: "Verankerung" der Zeiger-Ketten aller Niveaus an Head & Tail:

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 4

Größe einer perfekten Skip-Liste

- Listenhöhe (= max Höhe der Knoten) : $\log n$
- Anzahl der Zeiger pro Listenelement $\leq \log n$
- Gesamtzahl der Zeiger

$$\underbrace{[\log n] + 1}_{\text{Kopf}} + \sum_{i=0}^{\log n - 1} \frac{n}{2^i} \leq \log n + 1 + 2n$$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 5

Implementation von Skip-Listen

```

class SkipListNode(object):
    """Knotenklasse für Skip-Listen"""
    def __init__(self, key, height):
        self.key = key
        self.next = (height + 1) * [None]

class SkipList(object):
    """Implementiert eine Skip-Liste"""
    def __init__(self, maxHeight):
        self.maxHeight = maxHeight # = log n+1
        self.height = -1
        self.head = SkipListNode(-Infty, maxHeight)
        self.tail = SkipListNode(Infty, 0)
        for i in range(0, maxHeight):
            self.head.next[i] = self.tail
    
```

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 6

Suchen in Skip-Listen

- Wir suchen nach einem Key k in einer Skip-Liste wie folgt:
 - Wir beginnen "oben links" im Head der Skip-Liste
 - An der aktuellen Stelle p vergleichen wir k mit $y \leftarrow \text{key}(\text{next}(p) \text{ on same level})$
 - $k = y$: wir geben $\text{element}(\text{next}(p))$ zurück
 - $k > y$: wir machen "scan forward", d.h., $p \leftarrow \text{next}(p) \text{ on same level}$
 - $k < y$: wir machen "drop down", d.h., wir betrachten den nächst-niedrigeren Level
 - Falls der Algo versucht, unter Level 0 zu gehen, liefern wir NO_SUCH_KEY

erwartete Position von Key 16

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 7

```

def search( self, k ):
    """liefert den Knoten p in der Liste mit p.key = k,
    falls es ihn gibt, und None sonst"""
    p = self.head
    for i in range( height, -1, -1 ): # drop down
        # folge den Niveau-i-Zeigern
        while k > p.next[i].key:
            p = p.next[i] # scan forward
    # post-condition: p.key < k <= p.next[0].key
    p = p.next[0]
    if p.key == k:
        return p
    else:
        return None
    
```

Perfekte Skip-Listen:
 Suchen: $O(\log n)$ Zeit
 Einfügen / Entfernen: $\Omega(n)$ Zeit!

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 8

Randomisierte Skip-Listen

- Aufgabe der starren Verteilung der Höhen der Listenelemente
- Anteil der Elemente mit bestimmter Höhe wird beibehalten
- Höhen werden gleichmäßig und zufällig über die Liste verteilt
- Im folgenden: Begriff *Skip-Liste* bedeutet immer randomisierte Skip-Liste

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 9

Initialisierung

- Eine neue Liste wird wie folgt initialisiert:
 1. Ein Knoten *Head* wird erstellt und sein Key auf einen Wert gesetzt, der größer ist, als der größte Key, der in dieser Liste benutzt werden könnte ($+\infty$)
 2. Ein Knoten *Tail* wird erstellt, der Wert wird auf den kleinsten möglichen Key gesetzt ($-\infty$)
 3. Die Höhe einer solchen leeren Liste ist -1
 4. Alle Vorwärtszeiger des Heads zeigen auf Tail

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 10

Einfügen in randomisierte Skip-Listen

- Gegeben: neuer Key k
- 1. Suche (erfolglos) nach k ; die Suche endet bei dem Knoten q mit größtem Schlüssel, der kleiner als k ist (s. Such-Algo)
- 2. Füge neuen Knoten p mit Schlüssel k und zufällig gewählter Höhe nach Knoten q ein
 - Wähle dabei die Höhe von p so, daß für alle i gilt:

$$P[\text{Höhe von } p = i] = \frac{\#\text{Knoten der Höhe } i}{\#\text{Knoten}} = \frac{1}{2^{i+1}}$$
- 3. Sammle alle Zeiger, die über die Einfügestelle hinwegweisen, in einem Array und füge p in alle Niveau- i -Listen, mit $0 \leq i \leq \text{Höhe von } p$, ein

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 11

Beispiel

Einfügen von Schlüssel 10 mit Höhe 3

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 12

Python-Implementierung

```

def insert(self, k):
    """fügt den Schlüssel key in Skip-Liste ein"""
    # Suche den Schlüssel k und speichere in update[i] den
    # Zeiger auf Niveau i unmittelbar vor k
    update = self.height * [None]
    p = head
    for i in range(height, -1, -1):
        while p.next[i].key < k:
            p = p.next[i]
        update[i] = p
    p = p.next[0]
    if p.key == k:
        return # Schlüssel vorhanden
    newheight = randheight()
    if newheight > height: # Höhe der Skip-Liste anpassen
        for i in range(height+1, newheight+1):
            update[i] = self.head
        self.height = newheight
    p = SkipListNode(k, newheight)
    for i in range(0, newheight+1):
        # füge p in Niveau i nach update[i] ein
        p.next[i] = update[i].next[i]
        update[i].next[i] = p

```

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 13

Erzeugen zufälliger Höhen

```

def randheight(self):
    """ liefert eine zufällige Höhe zwischen 0 und
    maxHeight """
    height = 0
    # random() liefert zufällige Zahl aus [0,1]
    while random() <= 1/2 and height < maxHeight:
        height += 1
    return height

```

Es gilt:

$$P[\text{randheight} = i] = \frac{1}{2^{i+1}}$$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 14

Entfernen eines Schlüssels

- Gegeben: Schlüssel k

- Suche (erfolgreich) nach k
- Entferne Knoten p mit p.key = k aus allen Niveau-i-Listen mit $0 \leq i \leq \text{Höhe von } p$, und adjustiere ggfs. die Listenhöhe

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 15

```

def delete(self, k):
    """entfernt den Schlüssel k aus der Skip-Liste"""
    p = self.head
    update = self.height * [None]
    for i in range(height, -1, -1):
        # folge den Niveau-i Zeigern
        while p.next[i].key < k:
            p = p.next[i]
        update[i] = p
    p = p.next[0]
    if p.key != k:
        return # Schlüssel nicht vorhanden
    for i in range(0, len(p.next)):
        # entferne p aus Niveau i
        update[i].next[i] = p.next[i]
    # Passe die Höhe der Liste an
    while self.height >= 0 and self.head.next[height] == self.tail:
        self.height -= 1

```

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 16

Demo

SkipList Demonstration (C) 1997 Thomas Wenger

Level of this SkipList is 7. Maximum would be 7. Probability is 0.5.

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 17

Die erwartete Höhe einer Skip-Liste

- Zähle die Höhe hier ab 1
- Die Wahrscheinlichkeit, daß `randomheight()` den Wert `h` liefert, ist gleich der Wahrscheinlichkeit, daß `h-1` mal $\leq 1/2$ und dann $> 1/2$ auftritt
- Also ist die Wahrscheinlichkeit

$$P[\text{randomheight}() = h] = \left(\frac{1}{2}\right)^h$$
- Damit gilt für die erwartete Höhe $E[h]$ eines Knotens:

$$E[h] = P[h = 1] \cdot 1 + P[h = 2] \cdot 2 + \dots$$

$$= \sum_{h=1}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = 2$$

```

height = 1
while random() <= 1/2 and height < maxHeight:
    height += 1
return height
  
```

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 18

- Lemma:**
Die erwartete Höhe $H(N)$ einer Skip-Liste für N Elemente ist $\sim \log(N)$.
- Beweis:**
 - Sei $N(h)$ die Anzahl der Elemente, die Höhe $\geq h$ haben, in einer Skip-Liste mit n Elementen
 - Sei $\bar{N}(h)$ der erwartete Wert für $N(h)$ (= Mittelwert, gemittelt über viele Skiplisten)
 - $\bar{N}(0) = \frac{n}{2^0}$ $\bar{N}(1) = \frac{n}{2^1}$ $\bar{N}(2) = \frac{n}{2^2}$
 - $\bar{N}(\lfloor \log n \rfloor + 1) = \frac{n}{2^{\log n + 1}} = \frac{n}{2 * n} < 1$
 - Also: $H(n) \leq \lfloor \log(n) \rfloor + 1$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 19

Analyse der Suche

- Zur leichteren gedanklichen Handhabung betrachten wir die Skip-Liste als Folge von (übereinander angeordneten) Listen S_i , wobei $S_{i+1} \subset S_i$. Jeder Knoten hat Zeiger nach links, rechts, unten, und ggf. oben

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 20

- Annahme: für Analyse nehmen wir an, dass `randheight()` Werte aus, $0, \dots, +\infty$ produzieren darf (weil z.B. Wahrscheinlichkeit für `randheight() > 15` schon "praktisch" = 0 ist)
- Erinnerung: $P[\text{Knoten hat Höhe} \geq i] = \frac{1}{2^i}, i \geq 0$
- Suche startet links oben
 - Folge Niveau- i -Zeiger von einem Element zum nächsten, oder
 - steige um ein Niveau ab
- Ende, wenn bei Niveau 0 angekommen, und
 - Key gefunden, oder
 - Element.Key < Search-Key < Next-Element.Key
- Ziel: mittlere (=erwartete) Länge des Suchpfades berechnen

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 21

- Geniale Idee: Suchpfad rückwärts verfolgen
 - Starte mit Niveau-0-Element, an dem Pfad endet
- Allgemeinere Annahme:
 - wir befinden uns innerhalb irgend eines Elements p auf Niveau i
 - Liste ist nach links unbegrenzt

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 22

- Allgemeinere Frage: wie lang ist der Pfad von irgend einem Knoten p aus nach links im Mittel, wenn der Pfad dabei k Niveaus aufsteigt?
- Pfad besteht aus Elementstücken
 - 1 x Zeiger nach links verfolgen (selbes Niveau)
 - 1 x aufsteigen (selbes Element)

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 23

- "Rückwärts"-Algorithmus:


```

p = CurrentNode level 0 Pointer
while p != Head:
  if up(p) exists:
    p = up(p)
  else:
    p = left(p)
      
```
- Wahrscheinlichkeit, daß `up(p)` existiert, beträgt 0.5 (s. Funktion `randomheight()`)
- Aus diesem Grund erwarten wir, jeden Zweig der Schleife gleich oft zu durchlaufen

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 24

- Sei $h(p)$ Höhe von Element p
- Annahme war: $h(p) \geq i$
 - `randheight` würfelt und setzt mit Wahrscheinlichkeit $\frac{1}{2}$ die Höhe um 1 hoch, bzw. mit Wahrscheinlichkeit $\frac{1}{2}$ die Höhe auf genau i
 - Also 2 Fälle, beide haben Wahrscheinlichkeit $\frac{1}{2}$
- 1. Fall: $h(p) = i$
 - Suchpfad geht von p nach links zu Element p' auf Niveau i , $h(p') \geq i$
 - Pfad muß immer noch k Niveaus hochklettern
- 2. Fall: $h(p) > i$
 - Suchpfad klettert nun mind. 1 Niveau im Element p hoch
 - Pfad muß dann nur noch $k-1$ Niveaus hoch

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 25

- Bezeichne $EL(k)$ die erwartete Pfadlänge von p aus nach links und k Niveaus aufsteigend
- Rekursionsgleichung für $EL(k)$:

$$EL(k) = \underbrace{\frac{1}{2}(1 + EL(k))}_{\text{Fall 1}} + \underbrace{\frac{1}{2}(1 + EL(k-1))}_{\text{Fall 2}}$$
- $\Rightarrow EL(k) = 2 + EL(k-1)$
- Anfangsbedingung: $EL(0) = 0$
- Also: $EL(k) = 2k$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 26

- Jetzt mittlere (= erwartete) Länge eines Suchpfades in Skip-Liste der Länge N abschätzen
- Jetzt: Pfad startet "rechts unten"
 - Zerlege Pfad in 2 Teile
 - Teil 1: starte auf Niveau 0, steige $\log N$ Niveaus auf \rightarrow erwartete Länge dafür ist $2 \log N$
 - Teil 2: sei $M = \#$ Knoten mit Höhe $\geq \log N$
 - Wie groß ist erwarteter Wert für M ?
 - $E[M] = N \cdot P[\text{Knoten hat Höhe} \geq \log N] = N \cdot \frac{1}{2^{\log N}} = 1$
 - ist der Suchpfad auf Niveau $\log N$ hochgestiegen, muß er noch höchstens 1 Zeiger bis zum Kopf-Element verfolgen (im Mittel über alle Skip-Listen der Länge N)
 - Also: erwartete (= mittlere) Suchpfadlänge ist $2 \log N + 1$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 27

Zusammenfassung der Kosten in Skip-Listen

- Suchen dauert im Mittel $\sim 2 \log N$
- Ähnlich für Einfügen und Löschen
 - Übungsaufgabe

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 28

Häufigkeitsgeordnete Listen

- Bisherige Annahme beim Suchen in einer Liste:
 - Alle Suchanfragen **gleich häufig**, d.h. m.a.W.
 - Sei Key $K \in \mathcal{K}$
 - Annahme bisher war: $P[K = k_i] = \frac{1}{|\mathcal{K}|}$
 - Mittlerer (erwarteter) Aufwand für Suchen: $\frac{N}{2} \in O(N)$
- In der Praxis oft: **80/20-Regel**
 - 80% der Zugriffe erfolgen auf 20% der Daten
 - von diesen 80% betreffen wiederum 80% (64% der gesamten Zugriffe) 20% von den ersten 20% (4% aller Daten) usw.
 - Folge: wenn Liste nach diesen Häufigkeiten sortiert ist, ist mittlerer Aufwand zum Suchen nur noch 0.122N (ca. Faktor 8 besser)

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 29

Mittlere Kosten für Suchen bei bekannter Zugriffshäufigkeit:

- Das Array A enthalte die Daten A_1, \dots, A_n
- Sei $p_i = P[\text{gesuchter Key } K = A_i] = \frac{\text{Anzahl Suchanfragen mit Key } A_i}{\text{Anzahl Suchanfragen insgesamt}}$
- Vereinfachende Annahme: Key $K \in A$
- Kosten für (erfolgreiche) Suche nach Key A_i ist genau i
- Dann sind die mittleren Kosten für 1 Zugriff $\bar{C}(n) = 1 \cdot p_1 + \dots + n \cdot p_n$
- Wie minimiert man $\bar{C}(n)$?
- Lösung: ordne die A_i nach **Häufigkeit**, so daß also $p_1 \geq p_2 \geq \dots \geq p_n$

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 30

Selbstorganisierende Listen

- Problem:
 - die (relativen) Häufigkeiten der Suchanfragen sind **nicht im voraus** bekannt
 - die Häufigkeiten können sich im Lauf der Zeit **ändern**
- Lösung: baue die Liste zur Laufzeit um, basierend auf der Historie der bis dahin "gesehenen" Suchanfragen
- Genauer:
 - Gegeben: Liste A und Sequenz S von Suchanfragen s_0, \dots, s_m
 - Algorithmus bekommt ein s_j nach dem anderen, kann **nicht** in die Zukunft schauen
 - Algo bekommt nach jeder Suche s_j die Gelegenheit, die Liste A umzubauen
- Frage: **welche Umbaustrategie** ist die beste?

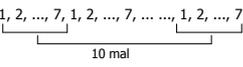
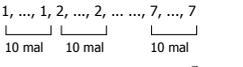
G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 31

Umbau-Strategien

- FC-Regel (Frequency count):**
 - Führe Häufigkeitszähler für jedes Element
 - Jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1
 - Sortiere Liste nach Häufigkeitszähler in gewissen Abständen
- T-Regel (Transpose):**
 - das Zielelement eines Suchvorgangs wird mit dem unmittelbar vorangehenden Element vertauscht
 - häufig referenzierte Elemente wandern (langsam) an den Listenanfang
- MF-Regel (Move-to-Front):**
 - Zielelement eines Suchvorgangs wird nach jedem Zugriff direkt an die erste Position der Liste gesetzt
 - relative Reihenfolge der übrigen Elemente bleibt gleich
 - in jüngster Vergangenheit referenzierte Elemente sind am Anfang der Liste (Lokalität kann gut genutzt werden)

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 32

Beispiel für MF-Regel

- Gegeben Liste $L = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$
- Zugriffsfolge $s: 1, 2, \dots, 7, 1, 2, \dots, 7, \dots, 1, 2, \dots, 7$

- Mittlere Kosten für 1 Zugriff bei Folge $s: \frac{\sum_{i=1}^7 i + 7 \cdot 9 \cdot 7}{10 \cdot 7} = 6.7$
- Zugriffsfolge $s: 1, \dots, 1, 2, \dots, 2, \dots, 7, \dots, 7$

- Mittlere Kosten für 1 Zugriff bei Folge $s: \frac{\sum_{i=1}^7 i + 9 \cdot 7 \cdot 1}{10 \cdot 7} = 1.3$
- Fazit:** MF kann besser sein als jede statische Anordnung, besonders, wenn die Zugriffe gehäuft auftreten

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 33

Beobachtungen

- MF-Regel ist "radikal", ist T-Regel evtl. besser?
- T-Regel ist "vorsichtiger", aber:
 - Betrachte $L = \langle 1, 2, \dots, N-1, N \rangle$ und die Zugriffsfolge $N, N-1, N, N-1, \dots$
 - T-Regel vertauscht immer die letzten beiden Elemente
 - Resultat: die durchschnittlichen Zugriffskosten sind hier N !
- FC-Regel
 - benötigt zusätzlichen Speicherplatz
 - Sortieren kostet mindestens Zeit $\sim N \rightarrow$ Zugriffskosten sind wieder $O(N)$
- Experimentelle Resultate zeigen:
 - T schlechter als FC,
 - MF und FC ungefähr gleich gut
 - MF hat Vorteile

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 34

Wichtiger Satz zur *Move-to-Front*-Regel

- Sei A irgend ein anderer Algorithmus zur Selbstanordnung — er darf sogar die gesamte Sequenz von Suchanfragen, S , im Voraus kennen
- Sei $C_{MF}(S)$ die Kosten für die Abarbeitung der Suchanfragen aus S gemäß MF-Regel (also Kosten für Suche selbst und Kosten für Umbau der Liste gemäß MF)
- Analog $C_A(S)$
- Satz [Sleator & Tarjan, 1985]:

$$C_{MF}(S) \leq 2C_A(S)$$
- Fazit:**
 - MF-Regel ist höchstens doppelt so schlecht wie irgend eine andere Regel
 - MF-Regel ist also nur um **kleinen Faktor schlechter als die optimale Regel!**
 - Selbst Vorkenntnisse über S helfen nicht viel!

G. Zachmann Informatik 2 - SS 06 Skip-Listen & selbst-org. Listen 35