

# ADS: Algorithmen und Datenstrukturen

Teil  $[\pi]$

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for  
Bioinformatics, **University of Leipzig**

3. November 2009

# Verkettete Listen

- Sequentielle Speicherung erlaubt schnelle Suchverfahren
  - falls Sortierung vorliegt
  - da jedes Element über Indexposition direkt ansprechbar
- Nachteile der sequentiellen Speicherung
  - hoher Änderungsaufwand durch Verschiebekosten:  $O(n)$
  - schlechte Speicherplatzausnutzung
  - inflexibel bei starkem dynamischem Wachstum
- Abhilfe: verkettete lineare Liste (Kette)
- Spezielle Kennzeichnung erforderlich für
  - Listenanfang (Anker)
  - Listenende
  - leere Liste

# Verkettete Liste: Implementierung

- Listenanfang wird durch speziellen Zeiger *head* (Kopf, Anker) markiert
- Leere Liste: *head* = null
- Listenende: *next*-Zeiger = null

Element: [*key* | ⟨*data*⟩ | *next*]

Liste = *head*-Zeiger auf das erste Element

Ende = *next*-Zeiger = null-pointer  $\emptyset$

Schematisch

head  $\rightarrow$  [7||]  $\rightarrow$  [13||]  $\rightarrow$  [34||]  $\rightarrow$  ...  $\rightarrow$  [90||]  $\rightarrow$  [9|| $\emptyset$ ]

# Verkettete Liste: Löschen

... [k<sub>1</sub>||n<sub>1</sub>] → [k<sub>2</sub>||n<sub>2</sub>] → [k<sub>3</sub>||n<sub>3</sub>] → ...  
... [k<sub>1</sub>||n<sub>2</sub>] → [k<sub>3</sub>||n<sub>3</sub>] → ...

```
if ( z1 != null AND z1 -> next != null )  
    z1 -> next = z1 -> next -> next
```

# Verkettete Liste: Suchen

1. (Initialisieren): Setze `aktuelles_Element := head`
2. (Test): Gesuchtes Element hier gefunden? Falls ja, return
3. (Abbruch?): `aktuelles_Element = Listenende`? Falls ja, re
4. (Iteration): Setze `aktuelles_Element := n{"a"}chstes_Ele`

Nur sequentielle Suche möglich (sowohl im geordneten als auch im ungeordneten Fall)! **Nachteile:**

- Einfügen und Löschen eines Elementes mit Schlüsselwert  $x$  erfordert vorherige Suche
- Bei Listenoperationen müssen Sonderfälle stets abgeprüft werden (Zeiger auf Null prüfen etc.)

**Fragen:** Wie funktioniert Löschen eines Elementes an Position (Zeiger)  $p$ ? Wie funktioniert Hintereinanderfügen von 2 Listen?

# Verkettete Liste: Suchen

Suche `key = x`:

Beginn: `Zeiger = head`.

```
while (Zeiger -> key != x) Zeiger = Zeiger -> next;
```

... Abbruch.

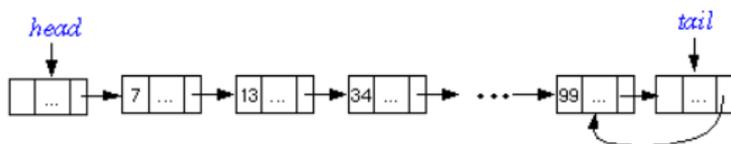
(Das gesuchte Element soll enthalten sein.)

# Verkettete Listen: Alternative Implementierung

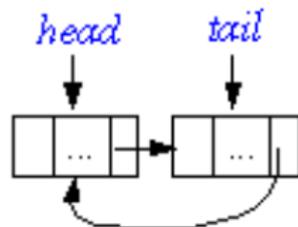
Dummy Elemente: `head` und `tail`

`tail` → `next` zeigt auf das letzte echte Listenelement

- Next-Zeiger des Dummy-Elementes am Listenende verweist auf vorangehendes Element (erleichtert Hintereinanderfügen zweier Listen und Abprüfen von Sonderfällen)



Leere Liste



`head` → `next` → `next` = `head`

# Doppelt gekettete Liste

`head`, `tail`, Zeiger nicht nur zum Nachfolger *next*, sondern auch zum Vorgänger *previous*

## Bewertung

- höherer Speicherplatzbedarf als bei einfacher Verkettung
- Aktualisierungsoperationen etwas aufwendiger (Anpassung der Verkettung)
- Suchaufwand in etwa gleich hoch, jedoch ggf. geringerer Suchaufwand zur Bestimmung des Vorgängers (Operation  $\text{PREVIOUS}(L, p)$ )
- geringerer Aufwand für Operation  $\text{DELETE}(L, p)$

Flexibilität der Doppelverkettung besonders vorteilhaft, wenn Element gleichzeitig Mitglied mehrerer Listen sein kann (Multilist-Strukturen)

# Vergleich verketteter Listen

- Suchaufwand bei ungeordneter Liste
  - erfolgreiche Suche:  $C_{avg} = (n + 1)/2$  (Standardannahmen: zufällige Schlüsselauswahl; stochastische Unabhängigkeit der gespeicherten Schlüsselmenge)
  - erfolglose Suche: vollständiges Durchsuchen aller  $n$  Elemente
- Einfügen oder Löschen eines Elements
  - konstante Kosten für Einfügen am Listenanfang
  - Löschen verlangt meist vorherige Suche
  - konstante Löschkosten bei positionsbezogenem Löschen und Doppelverkettung
- Sortierung bringt kaum Vorteile
  - erfolglose Suche verlangt im Mittel nur noch Inspektion von  $(n + 1)/2$  Elementen (Bei Abbruch der Suche mit gleicher Wahrscheinlichkeit an allen Positionen der Liste)
  - lineare Kosten für Einfügen in Sortierreihenfolge

# Skip-Listen

**Ziel:** verkettete Liste mit logarithmischem Aufwand für Suche, Einfügen und Löschen von Schlüsseln (Wörterbuchproblem)

Verwendung sortierter verketteter gespeicherter Liste mit zusätzlichen Zeigern

## Prinzip

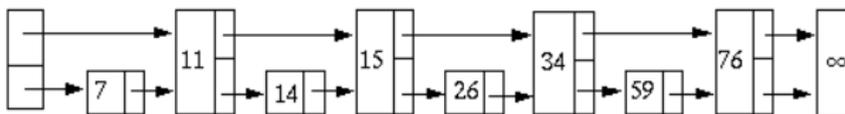
- Elemente werden in Sortierordnung ihrer Schlüssel verkettet
- Führen mehrerer Verkettungen auf unterschiedlichen Ebenen:

Verkettung auf Ebene 0 verbindet alle Elemente;

Verkettung auf Ebene 1 verbindet jedes zweite Element;

...

Verkettung auf Ebene  $i$  verbindet jedes  $2^i$ -te Element ( $i = 0, 1, \dots$ )



# Perfekte Skip-Liste - Anzahl Zeiger & Kosten

Zahl der Zeiger auf eigentliche Elemente:  $n = 2^k$

Ebene 0:  $n$  Zeiger

Ebene 1:  $n/2$  Zeiger

...

Ebene  $k$ : 1

Insgesamt  $n + n/2 + n/4 + \dots + 1 = \sum_{i=0}^k \frac{n}{2^i} \geq 2n$

Anzahl der Ebenen (Listenhöhe):  $1 + \lfloor \log_2 n \rfloor$

maximale Anzahl der Zeiger pro Element  $\geq \lfloor \log_2 n \rfloor$

maximale Gesamtzahl der Zeiger  $< 2n$

Suche  $O(\log n)$

Perfekte Skip-Listen zu aufwendig bezüglich Einfügungen und Löschvorgängen (vollständige Reorganisation erforderlich, Kosten  $O(n)$ )

# Randomisierte Skip-Listen

- Strikte Zuordnung eines Elementes zu einer Höhe (Anzahl der Ebenen) wird aufgegeben
- Höhe eines neuen Elementes  $x$  wird nach Zufallsprinzip ermittelt, jedoch so, daß die relative Häufigkeit der Elemente pro Ebene (Höhe) eingehalten wird, d.h.  $P(i) = 1/2^i$  (für Höhen  $i = 1, 2, \dots$ )
- Somit entsteht eine “zufällige” Struktur der Liste
- Kosten für Einfügen und Löschen im wesentlichen durch Aufsuchen der Einfügeposition bzw. des Elementes bestimmt:  $O(\log N)$

# Stacks als spezielle Listen

Synonyme: Stapel, Keller, LIFO-Liste usw.

Stack kann als spezielle Liste aufgefaßt werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt, vorgenommen werden Stack-Operationen (ADT):

- 1 CREATE: Erzeugt den leeren Stack
- 2 INIT( $S$ ): Initialisiert  $S$  als leeren Stack
- 3 PUSH( $S, x$ ): Fügt das Element  $x$  als oberstes Element von  $S$  ein
- 4 POP( $S$ ): Löschen des Elementes, das als letztes in den Stack  $S$  eingefügt wurde
- 5 TOP( $S$ ): Abfragen des Elementes, das als letztes in den Stack  $S$  eingefügt wurde
- 6 EMPTY( $S$ ): Abfragen, ob der Stack  $S$  leer ist

Alle Operationen mit konstanten Kosten realisierbar:  $O(1)$

## Beispiel: Operationen auf Stack

$S = |a, b$

POP( $S$ )

$S = |a$

PUSH( $S, c$ )

$S = |a, c$

EMPTY( $S$ ) ... false

# Anwendung: Wohlgeformte Klammerausdrücke

## Definition

- $()$  ist ein wohlgeformter Klammerausdruck (wgK)
- Sind  $w_1$  und  $w_2$  wgK, so ist auch ihre Konkatenation  $w_1 w_2$  ein wgK
- Mit  $w$  ist auch  $(w)$  ein wgK
- Nur die nach den vorstehenden Regeln gebildeten Zeichenreihen bilden wgK
- Beispiel:  $(( ( ) ) ( ) )$  ist wohlgeformt.

Algorithmus: Idee: Stapel zur Speicherung öffnender Klammern.

- Einlesen des Ausdrucks von links nach rechts.
- Wird eine öffnende Klammer gelesen: Speichere diese im Stack
- Wird eine schließende Klammer gelesen: Entferne das oberste Stack-Element.
- wgK liegt vor, wenn Stack am Ende leer ist

# Anwendung: UPN-Notation

dc - an arbitrary precision calculator

$(a+b) * (c+d/e) \implies a b + c d e / + *$

→TAFEL

# Schlangen

Synonyme: FIFO-Schlange, Warteschlange, Queue  
Spezielle Liste, bei der die Elemente an einem Ende (hinten) eingefügt und am anderen Ende (vorne) entfernt werden  
Operationen:

- CREATE: Erzeugt die leere Schlange
- INIT (Q): Initialisiert Q als leere Schlange
- ENQUEUE (Q, x): Fügt das Element x am Ende der Schlange Q ein
- DEQUEUE (Q): Löschen des Elementes, das am längsten in der Schlange verweilt (erstes Element)
- FRONT (Q): Abfragen des ersten Elementes in der Schlange
- EMPTY (Q): Abfragen, ob die Schlange leer ist

$Q = [a, b, c]$       ENQUEUE(Q, d)       $Q = [a, b, c, d]$   
DEQUEUE(Q)       $Q = [b, c, d]$

# Vorrangwarteschlangen

(priority queues)

Jedes Element erhält Priorität. Entfernt wird stets Element mit der höchsten Priorität (Aufgabe des FIFO-Verhaltens einfacher Warteschlangen)

Operationen:

- CREATE: Erzeugt die leere Schlange
- INIT(P): Initialisiert P als leere Schlange
- INSERT(P, x): Fügt neues Element x in Schlange P ein
- DELETE(P): Löschen des Elementes mit der höchsten Priorität aus P
- MIN(P): Abfragen des Elementes mit der höchsten Priorität
- EMPTY(P): Abfragen, ob Schlange P leer ist.

Sortierung nach Prioritäten beschleunigt Operationen DELETE und MIN auf Kosten von INSERT.

9, 7, 4, 3

INSERT(P,5)

9, 7, 5, 4, 3

INSERT(P,1)