ADS: Algorithmen und Datenstrukturen

Teil $\lfloor \pi \rfloor$

Prof. Peter F. Stadler & Dr. Christian Höner zu Siederdissen

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

26.10.2015

[Letzte Aktualisierung: 26/10/2015, 14:08]

Bürokratie

- Prüfen Sie ihre Modulanmeldung! Anmeldung nur bis 01.11. möglich
- Checken Sie ihre Gruppenzuordnung:
- via BioInf / Teaching / Current / ADS I
- `www.bioinf.uni-leipzig.de/Leere/WS1516/ADS/punkt.php`
- nicht angemeldet? mail an `will@bioinf.uni-leipzig.de`: Namen, Matrikelnummer, Studienrichtung, Gruppenwunsch (ohne Anspruch)
- falsche Daten? ebenso!
- Alle An-/Um-meldungen ohne Anspruch
- Übungsgruppe 7 (Kianian) ist englischsprachig
- Übungszettel 1 ab heute auf der Homepage; Abgabe: spätestens 02.11. vor der Vorlesung

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
- Spezielle Kennzeichnung erforderlich für
  - Listenanfang (Anker)
  - Listenende
  - leere Liste

Verkettete Liste: Implementierung

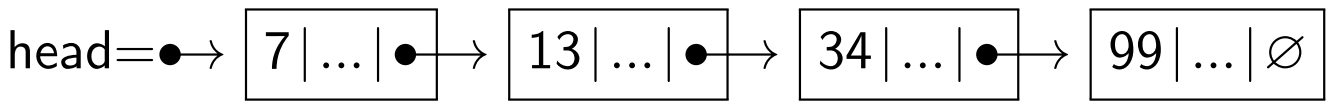
- Listenanfang wird durch speziellen Zeiger head (Kopf, Anker) markiert
- Leere Liste: *head* = `null`
- Listenende: *next*-Zeiger = `null`

Element: $\boxed{\text{key}\,|\,\text{data}\,|\,\text{next}}$

Listenanfang: *head*-Zeiger auf das erste Element

Listenende: *next*-Zeiger = `null`-pointer $\varnothing$

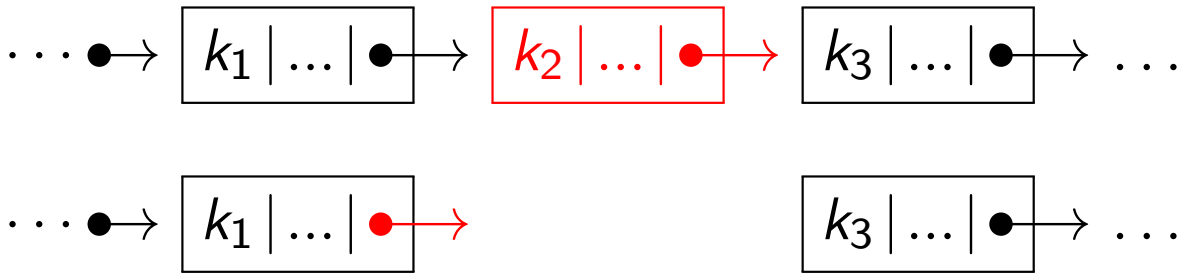
Beispiel für verkette Liste:



Anmerkung (Implementierung von Zeigern): jedes Listenelement liegt im Speicher an einer bestimmten Stelle, die durch eine ganze Zahl angegeben wird (=Adresse). Wert eines Zeigers = Adresse = Zahl: z.B.

head=64; ... 32: $\boxed{13\,|\,\ldots\,|\,50}$ ... 42: $\boxed{99\,|\,\ldots\,|\,0}$ ... 50: $\boxed{34\,|\,\ldots\,|\,42}$ ... 64: $\boxed{7\,|\,\ldots\,|\,32}$ ...

Verkettete Liste: Löschen



Lösche das Element hinter dem Element, auf das der Zeiger p zeigt:

```
if ( p != null AND p→next != null )
    p→next = p→next→next
```

Verkettete Liste: Suchen

```
1. (Initialisieren): Setze p := head
2. (Test): Gesuchtes Element hier gefunden?
   Falls ja, return("gefunden")
3. (Abbruch?): p = Listenende?
   Falls ja, return("nicht gefunden").
4. (Iteration): Setze p := nächstes_Element.
   Gehe zu 2.
```

Nur sequentielle Suche möglich (ob geordnet oder ungeordnet)!

Aber: Einfügen und Löschen eines Elementes mit Schlüsselwert x erfordert vorherige Suche.

Wie funktioniert Löschen eines Elementes auf das ein Zeiger p zeigt? Wie funktioniert Verketten ("Hintereinanderfügen") von 2 Listen ?

Verkettete Liste: Suchen

```
SEARCH( Schlüssel x, Liste L ) {
  Zeiger p=L.head;
  while (p→key != x) { p = p→next; }
  return p;
}
```

Wir nehmen hier an, dass das gesuchte Element enthalten ist. Was gibt SEARCH(x,L) zurück?

Was passiert, wenn kein Element mit Schlüssel x vorkommt? Was sollte an dem Code geändert werden?

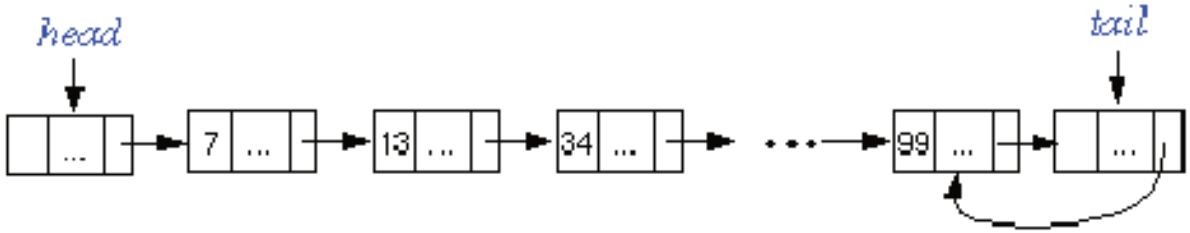
Bisheriger Nachteil von Listen: Bei Listenoperationen müssen viele Sonderfälle abgeprüft werden (Zeiger auf Null prüfen etc.)

Alternativ: Liste mit Kopf- und Schwanzzeiger

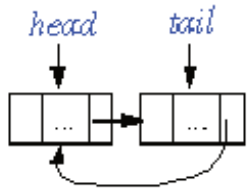
Dummy Elemente: `head` und `tail`

(ausserdem: L.head und L.tail zeigen auf diese Elemente)

- `tail` $\rightarrow$ *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)



Implementierung "leere Liste":



[`head`$\rightarrow$`next`$\rightarrow$`next` = `head`]

Wie vereinfacht das Listenoperationen?
z.B VERKETTEN(L1,L2)

Doppelt verkettete Listen

Zeiger nicht nur zum Nachfolger *next*, sondern auch zum Vorgänger *previous*; (zusätzlich wieder Dummy-Elemente `head`, `tail`)

Eigenschaften

- Bestimmung des Vorgängers in konstanter Zeit ($p \rightarrow$ previous)
- höherer Speicherplatzbedarf als bei einfacher Verkettung
- Aktualisierungsoperationen etwas aufwendiger (Anpassung der Verkettung)
- Suchaufwand in etwa gleich hoch
- Flexibler: z.B. Löschen des Elements, auf das p zeigt, wird einfacher (Operation DELETE(p, L))

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

Zusammenfassung: Eigenschaften verketteter Listen

- Suchaufwand
  - erfolgreiche Suche: $C_{avg} = (n+1)/2$ (Standardannahmen: zufällige Schlüsselauswahl; stochastische Unabhängigkeit der gespeicherten Schlüsselmenge)
  - erfolglose Suche:
    - falls unsortiert: vollständiges Durchsuchen aller n Elemente
    - falls sortiert: nur noch Inspektion von $(n+1)/2$ Elementen (Bei Abbruch der Suche mit gleicher Wahrscheinlichkeit an allen Positionen der Liste)
- Einfügen / Löschen
  - konstante Kosten für Einfügen an Listenanfang (oder an bestimmter Position)
  - konstante Kosten für positionsbezogenes Löschen (bei Doppelverkettung)
  - lineare Kosten für schlüsselbezogenes Löschen
  - lineare Kosten für schlüsselbezogenes Einfügen, d.h. nach einem Element mit gegebenem Schlüssel; auch Einfügen in Sortierreihenfolge
- Verkettung: konstant bei Listen mit head und tail Zeigern

Skip-Listen

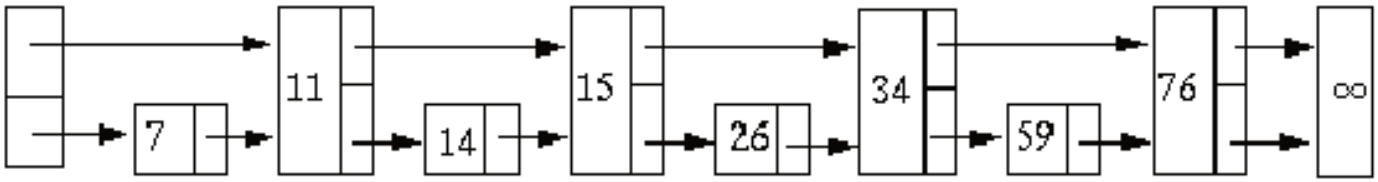
Ziel: Datenstruktur mit logarithmischem Aufwand für Suche, Einfügen und Löschen von Schlüsseln

Prinzip

Verwendung sortierter verkettet gespeicherter Liste mit zusätzlichen Zeigern

- Elemente werden in Sortierordnung ihrer Schlüssel verkettet
- Führen mehrerer Verkettungen auf 2 oder mehreren Ebenen:

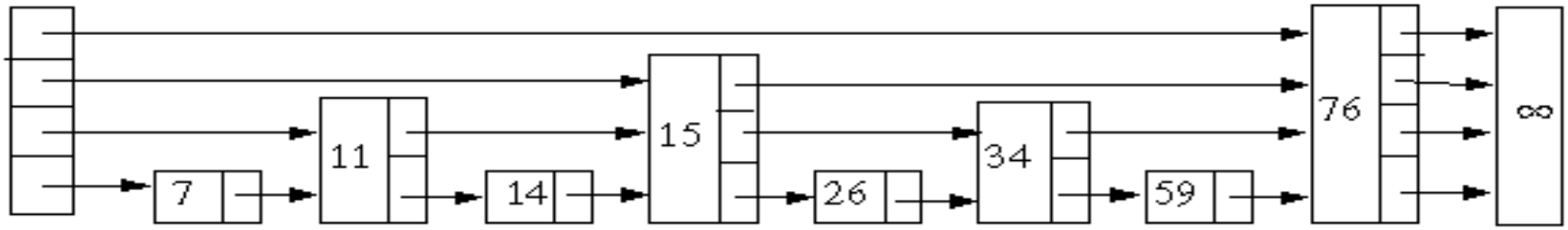
Verkettung auf Ebene 0 verbindet alle Elemente;
z.B. Verkettung auf Ebene 1 verbindet jedes zweite Element



Allgemein: ein Zeiger auf Ebene i zeigt auf 2^i-nächstes Element

Warum kann man jetzt schneller suchen als in einfacher Liste?

Perfekte Skip-Liste - Verkettung



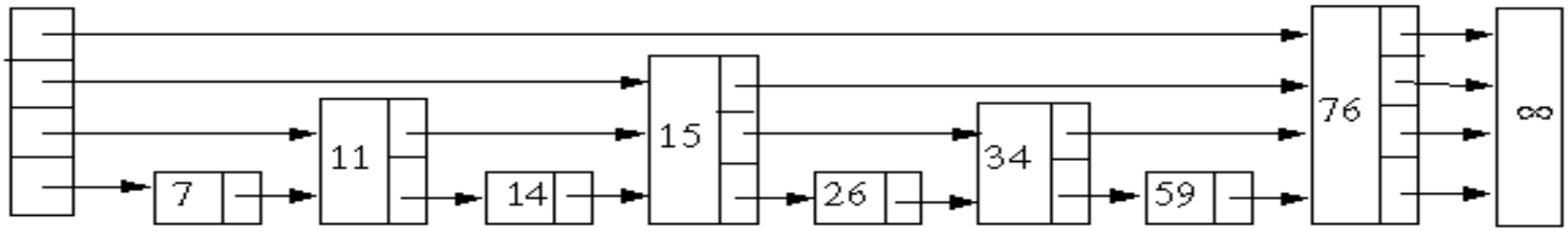
Zeiger:

(Annahme: Listenlänge $n = 2^k$)

Ebene 0: Zeiger an jeder Position (auf nächste P.) n
Ebene 1: Zeiger an jeder 2-ten Position (auf übernächste P.) $n/2$
Ebene 2: Zeiger an jeder 4-ten Position (auf 4.-nächste P.) $n/4$
...
Ebene k: Zeiger an erster Position (auf tail) 1

Insgesamt $n + n/2 + n/4 + \cdots + 1 = \sum_{i=0}^{k} \frac{n}{2^i} < 2n$ Zeiger

Perfekte Skip-Liste: weitere Eigenschaften & Suche



Anzahl der Ebenen $= 1 + \lfloor \log_2 n \rfloor$
Höhe eines Elements = Anzahl der Zeiger des Elements -1
maximale Höhe: $\lfloor \log_2 n \rfloor$
Gesamtzahl der Zeiger $< 2n$

Suche: ähnlich zu binärer Suche $\Rightarrow O(\log n)$

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

Randomisierte Skip-Listen

- Aufgeben der "Perfektion": keine starre Festlegung der Höhe eines Elementes nach seiner Position
- Höhe eines neuen Elementes x wird zufällig gewählt, aber mit Wahrscheinlichkeit jeder Höhe so wie in perfekter Skipliste, d.h. $P(h) = 1/(2 \cdot 2^h)$ (für Höhen $h = 0, 1, 2, ...$)
- 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

Abstrakter Daten Typ (ADT) **Stack** gegeben durch Operationen:

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: $S = |a, b$ POP(S) $S = |a$ PUSH(S, c) $S = |a, c$
EMTPY(S) $\longrightarrow$ 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 vorigen Regeln gebildeten Zeichenreihen bilden wgK

Beispiel: ((( )) ( )) ist wohlgeformt.

Algorithmus: "Erkennen von wgK"

Idee: Benutze Stapel zum Speichern öffnender Klammern.

- Einlesen des Ausdrucks von links nach rechts.
- Bei Lesen von öffnenden Klammer: Speichere diese auf Stack (PUSH)
- Lesen schließender Klammer: Entferne oberstes Stack-Element (POP)
- wgK liegt vor, wenn der Stack vor POP nie leer war und er am Ende leer ist (EMPTY)

Anwendung: Berechnung von Ausdrücken in Umgekehrt Polnischer Notation (UPN, Postfix-Ausdrücke)

UPN = Postfix-Notation arithmetischer Ausdrücke; erlaubt besonders einfache Auswertung mit Stack

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Auswertung:

- Lese Symbole in UPN von links nach rechts.
- Wenn Zahl: lege auf Stack.
- Wenn m-stelliger Operator: nehme *m* Operanden vom Stack, führe Operation aus und lege Ergebnis auf Stack.

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 1

`1 2 +`

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 2

3

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 3

`3 4 6 3 /`

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 4

`3 4 2`

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 5

`3 4 2 +`

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 6

3 6

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 7

`3 6 *`

Beispiel: `(1+2) * (4+6/3)` $\Longrightarrow$ `1 2 + 4 6 3 / + *`

Stack Schritt 8

18

Schlangen

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

Beispiel:

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

Vorrangwarteschlangen (Priority Queues)

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

Beispiel: P=[9, 7, 4, 3] INSERT(P,5) [9, 7, 5, 4, 3]
INSERT(P,1) [9, 7, 5, 4, 3, 1] DELETE(P) [7, 5, 4, 3, 1]

Zusammenfassung

- Verkettete Listen
  - dynamische Datenstrukturen mit geringem Speicheraufwand und geringem Änderungsaufwand
  - Implementierungen: einfach vs. doppelt verkettete Listen
  - hohe Flexibilität
  - hohe Suchkosten
- Skip-Listen
  - logarithmische Suchkosten
  - randomisierte statt perfekter Skip-Listen zur Begrenzung des Änderungsaufwandes
- ADT-Beispiele: Stack, Queue, Priority Queue
  - spezielle Listen mit eingeschränkten Operationen (LIFO bzw. FIFO)
  - formale ADT-Spezifikation zur Festlegung der implementierungsunabhängigen Eigenschaften
  - effiziente Implementierbarkeit der Operationen: O(1)
  - zahlreiche Anwendungsmöglichkeiten