

Algorithmen und Datenstrukturen 1

3. Vorlesung

Peter F. Stadler

Universität Leipzig
Institut für Informatik
studla@bioinf.uni-leipzig.de

3. Verkettete Listen, Stacks, Queues

Verkettete lineare Listen

- einfache Verkettung
- doppelt verkettete Listen
- Vergleich der Implementierungen

Fortgeschrittenere Kettenstrukturen

- Skip-Listen

Spezielle Listen: Stack, Queue, Priority Queue

- Operationen
- formale ADT-Spezifikation
- Anwendung

Verkettete Speicherung linearer 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)

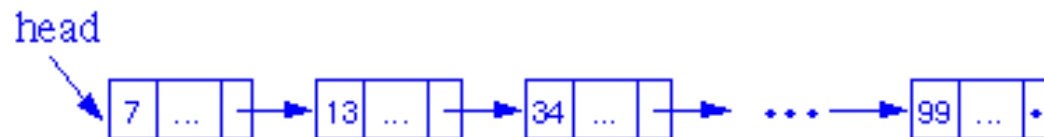
Spezielle Kennzeichnung erforderlich für

- Listenanfang (Anker)
- Listenende
- leere Liste

Verkettete Liste: Implementierung 1

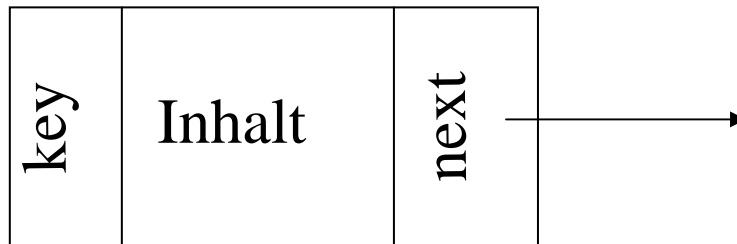
Implementierung 1:

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



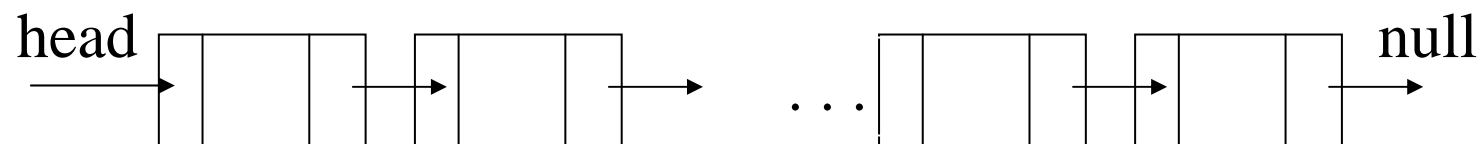
Verkettete Listen – Implementierung

Listenelement:



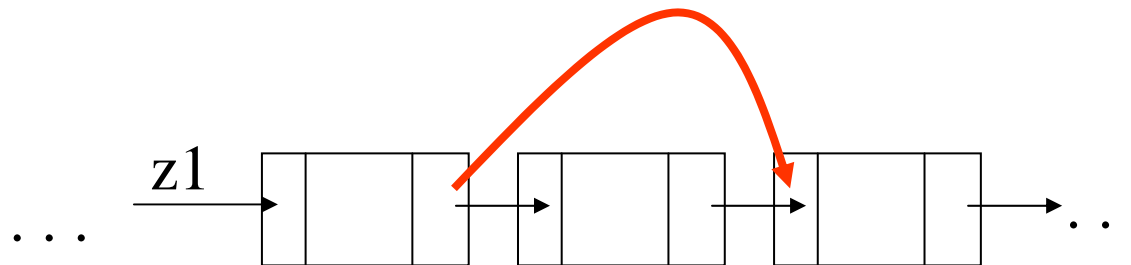
Liste = head – Zeiger auf das erste Listenelement

Listenende: Next-Zeiger = null – Zeiger



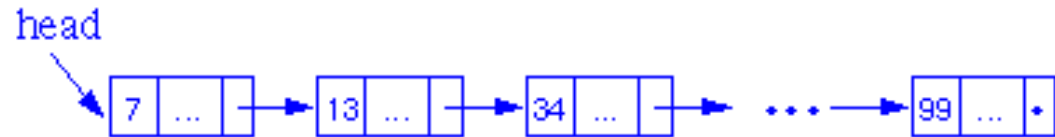
Verkettete Listen – Löschen

Löschen:



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

Verkettete Liste: Implementierung 1



Beispiel: Suchen von Schlüsselwert x

1. (Initialisieren): Setze `aktuelles_Element := head`
 2. (Test): Gesuchtes Element hier gefunden? Falls ja, `return(„gefunden“)`.
 3. (Abbruch?): `aktuelles_Element = Listenende?` Falls ja, `return(„nicht gefunden“)`.
 4. (Iteration): Setze `aktuelles_Element := nächstes_Element`. Gehe zu 2.
- D.h., 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 Listen – Suchen

Suche key = x:

Beginn: Zeiger = head.

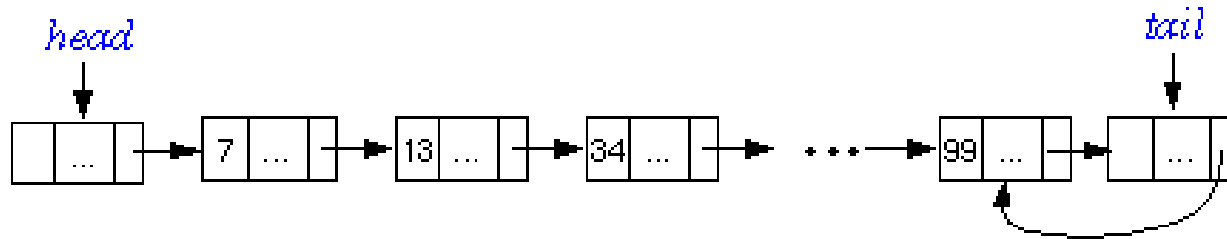
while (Zeiger -> key != x) Zeiger = Zeiger -> next;

Abbruch....

(Das gesuchte Element soll enthalten sein.)

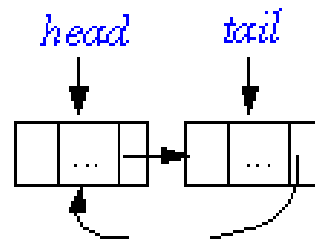
Verkettete Liste: Implementierung 2

Dummy- (uneigentliches) Element am Listenanfang sowie am Listenende (Zeiger head und tail)



- Next-Zeiger des Dummy-Elementes am Listenende verweist auf vorangehendes Element (erleichtert Hintereinanderfügen zweier Listen und Abprüfen von Sonderfällen)
- Die Liste ist durch die Zeiger head und tail gegeben.

- Leere Liste:

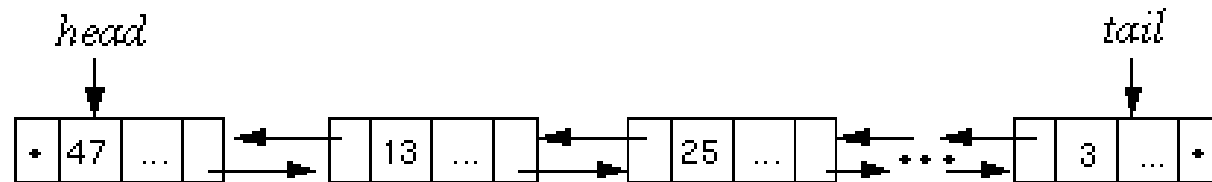


Leer falls

head -> next -> next = head

Verkettete Liste: Implementierung 3

Doppelt gekettete Liste



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 PREVIOUS (L, p))
- geringerer Aufwand für Operation 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: $C_{\text{avg}} = \frac{n+1}{2}$

(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

Vergleichstabelle

<i>VERGLEICH der 3 Implementierungen</i>	Implem. 1	Implem. 2	Implem. 3 (Doppelkette)
Einfügen am Listenanfang			
Einfügen an gegebener Position			
Löschen an gegebener Position			
Suchen eines Wertes			
Hintereinanderfügen von 2 Listen			

Erklärungen:

Position = Zeiger auf Listenelement bzw. auf next – Zeiger des Vorgängers
(implementierungsabhängig)

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

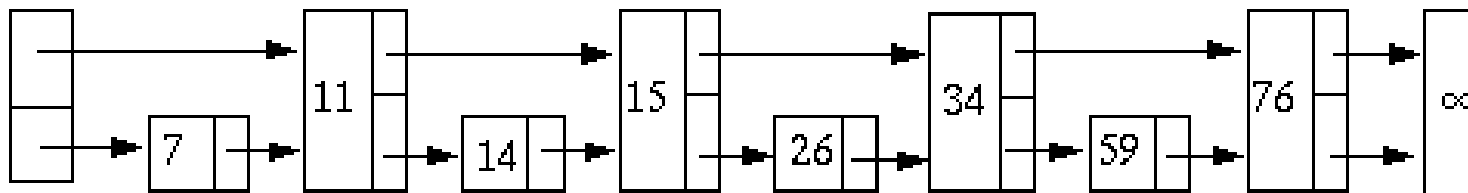
- Elemente werden in Sortierordnung ihrer Schlüssel verkettet
- Führen *mehrerer* Verkettungen auf unterschiedlichen Ebenen:

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

Zahl der Zeiger der eigentlichen Elemente : Sei $n = 2^k$. Zahl der Zeiger

Auf Ebene 0: n

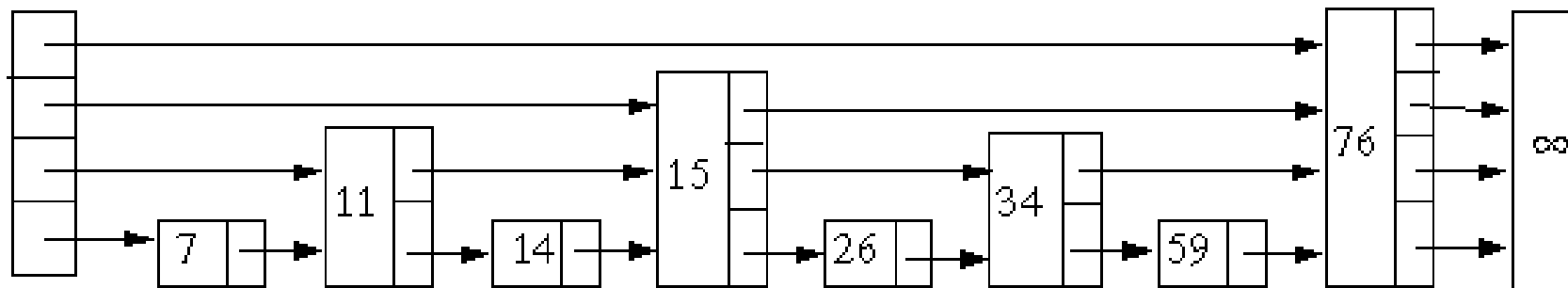
Auf Ebene 1: $n/2$

⋮

Auf Ebene $k = \log_2 n$: 1 (Zeiger von Element 1 auf tail - Element)

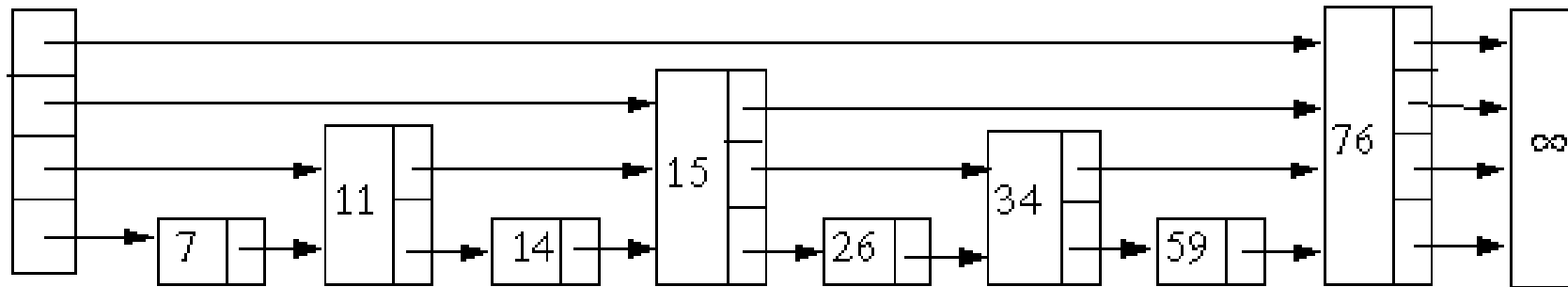
$$\text{Gesamtzahl: } n + \frac{n}{2} + \dots + 1 = \sum_{i=0}^k \frac{n}{2^i} \leq 2n$$

$$\text{denn } \sum_{i=0}^k \frac{1}{2^i} = 2 - 2^{-k} \text{ oder mit geometrischer Reihe: } \frac{1}{1-a} = 1 + a + a^2 + \dots \text{ und } a = \frac{1}{2}.$$



Perfekte Skip-Liste – Kosten

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



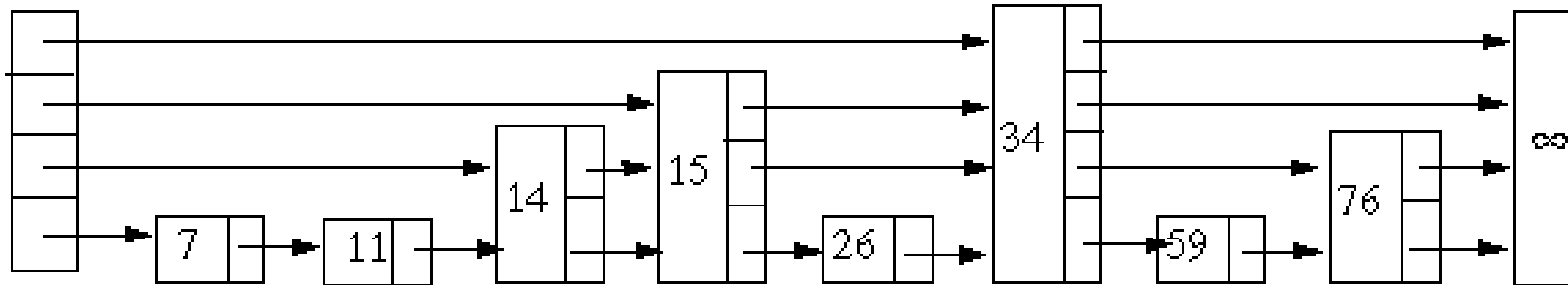
- Jedes eigentliche Element hat höchstens $\lfloor \log_2 n \rfloor$ Zeiger
- max. Gesamtanzahl der Zeiger $< 2n$ (ähnlich doppelt verketteter Liste)
- Suche: $O(\log n)$

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.
$$P(\text{Höhe von } x = i) = 1 / 2^i \text{ (für Höhen } i = 1, 2, \dots)$$
- 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):

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

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

Formale ADT-Spezifikation: Stacks

Formale ADT-Spezifikation zur Festlegung der implementierungsunabhängigen Stack-Eigenschaften

- ELEM = Wertebereich der Stack-Elemente
- STACK = Menge der Zustände, in denen sich der Stack befinden kann
- leerer Stack: $s_0 \in \text{STACK}$
- Stack-Operationen werden durch ihre Funktionalität charakterisiert. Ihre Semantik wird durch Axiome festgelegt.

Definitionen:

Datentyp STACK

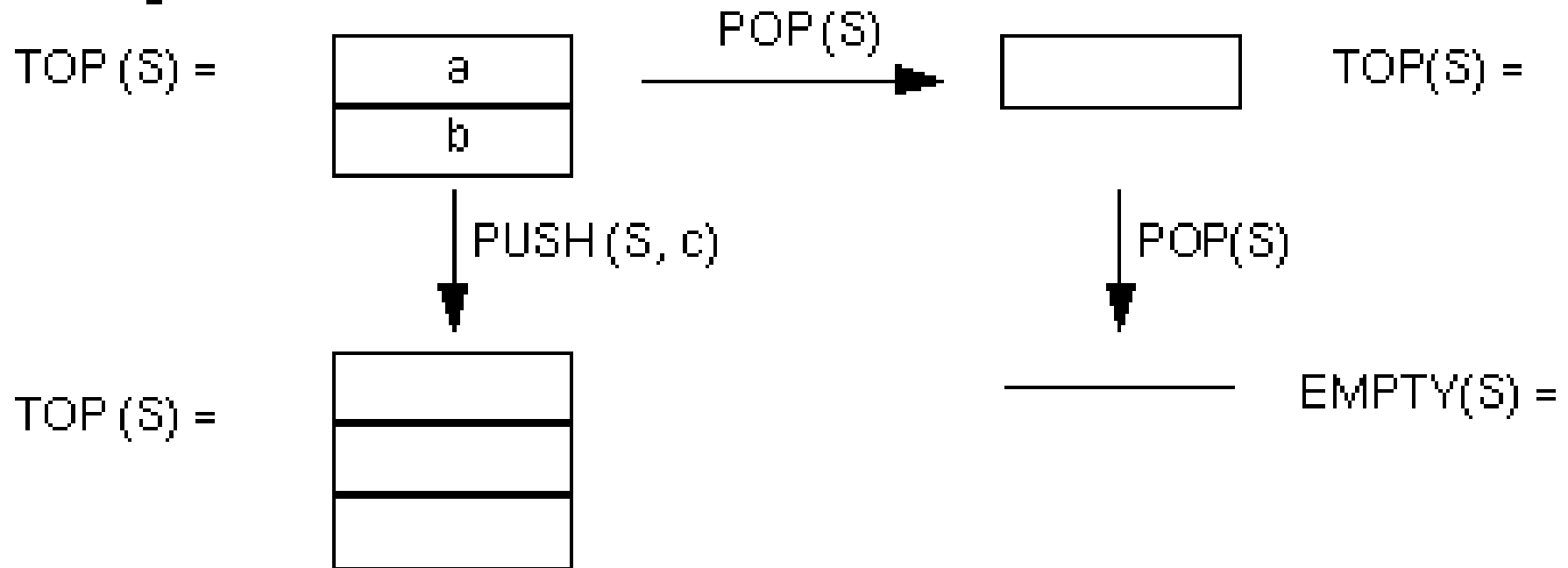
Basistyp ELEM

Operationen:

CREATE:		$\rightarrow \text{STACK};$
INIT:	STACK	$\rightarrow \text{STACK};$
PUSH:	STACK \times ELEM	$\rightarrow \text{STACK}$
POP:	STACK $\setminus \{s_0\}$	$\rightarrow \text{STACK};$
TOP:	STACK $\setminus \{s_0\}$	$\rightarrow \text{ELEM};$
EMPTY:	STACK	$\rightarrow \{\text{TRUE}, \text{FALSE}\}.$

Beispiel: Operationen auf Stack

Beispiel



Anwendungsbeispiel 1: Erkennen wohlgeformter 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. Ablauf des Algorithmus:

- 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

Anwendungsbeispiel 2: Berechnung von Ausdrücken in Umgekehrter Polnischer Notation (Postfix-Ausdrücke)

Beispiel: $(a+b) * (c+d/e) \Rightarrow a b + c d e / + *$

Lösungsansatz

- Lesen des Ausdrucks von links nach rechts
- Ist das gelesene Objekt ein Operand, wird es auf den STACK gebracht
- Ist das gelesene Objekt ein m-stelliger Operator, dann wird er auf die m obersten Elemente des Stacks angewandt. Das Ergebnis ersetzt diese m Elemente

Abarbeitung des Beispielausdrucks: (Der Stack wächst nach unten!)

UPN	a	b	+	c	d	e	/	+	x
Platz 1									
Platz 2									
Platz 3									
Platz 4									

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

Formale ADT-Spezifikation: Queue

formale ADT-Spezifikation

- ELEM = Wertebereich der Schlangen-Elemente
- QUEUE = Menge der möglichen Schlangen-Zustände
- leere Schlange: $q_0 \in \text{QUEUE}$

Datentyp: QUEUE

Basistyp: ELEM

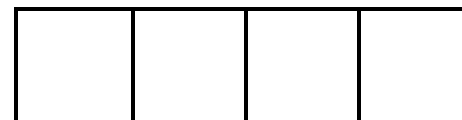
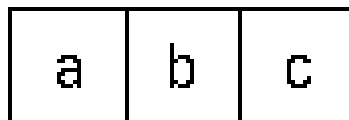
Operationen:

CREATE :		→ QUEUE;
INIT :	QUEUE	→ QUEUE;
ENQUEUE :	QUEUE × ELEM	→ QUEUE;
DEQUEUE :	QUEUE- $\{q_0\}$	→ QUEUE;
FRONT :	QUEUE- $\{q_0\}$	→ ELEM;
EMPTY :	QUEUE	→ {TRUE,FALSE}.

Beispiel: Operationen auf Queue

Beispiel:

FRONT (Q) =



ENQUEUE (Q,d)



DEQUEUE (Q)



FRONT (Q) =

DEQUEUE (Q)



FRONT (Q) =

Vorrangwarteschlangen (priority queues)

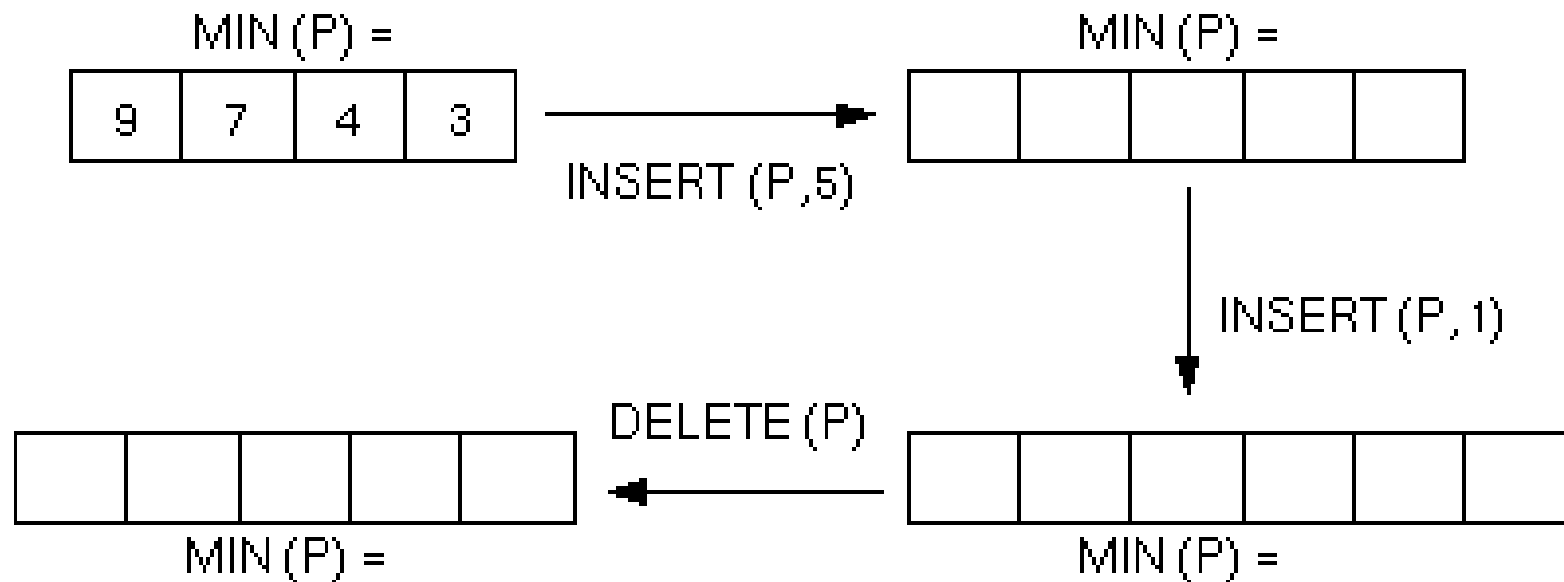
Jedes Element erhält Priorität. Entfernt wird stets Element mit der höchsten Priorität (Aufgabe des FIFO-Verhaltens einfacher Warteschlangen)

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.

Beispiel: Operationen auf PQueue



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