3. Verkettete Listen, Stacks, Queues

Verkettete lineare Listen
- Einfache Verkettung
- Doppelt verkettete Listen
- Vergleich der Implementierungen
- Iterator-Konzept

Fortgeschrittenere Kettenstrukturen
- Selbstorganisierende (adaptive) Listen
- 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

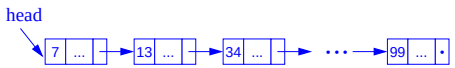
Nachteile der sequentiellen Speicherung
- hoher Änderungsaufwand durch Verschiebekosten: O(n)
- schlechte Speicherplatzausnutzung
- inflexibel bei starkem dynamischem Wachstum

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



Beispiel: Suchen von Schlüsselwert x

```
class KettenElement {
  int key;
  String wert;
  KettenElement next = null;
}
class KettenListe1 implements Liste {
  KettenElement head = null;
  ...
  public boolean search(int x) {
    KettenElement element = head;
    while ((element != null) && (element.key != x))
      element = element.next;
    return (element != null);
  }
}
```

Implementierung 1(Forts.)

nur sequentielle Suche möglich (sowohl im geordneten als auch im ungeordneten Fall) !

Einfügen und Löschen eines Elementes mit Schlüsselwert x erfordert vorherige Suche

Bei Listenoperationen müssen Sonderfälle stets abgeprüft werden (Zeiger auf Null prüfen etc.)

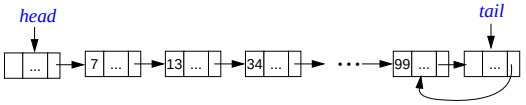
Löschen eines Elementes an Position (Zeiger) p ?

Hintereinanderfügen von 2 Listen ?

Verkette Liste: Implementierung 2

Implementierung 2:

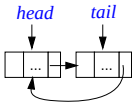
- Dummy-Element am Listenanfang sowie am Listenende (Zeiger *head* und *tail*)



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

```
class KettenListe2 implements Liste {
  KettenElement head = null;
  KettenElement tail = null;
  /** Konstruktor */
  public KettenListe2() {
    head = new KettenElement();
    tail = new KettenElement();
    head.next = tail;
    tail.next = head;
  }...}
```

- Leere Liste:



Implementierung 2 (Forts.)

Suche von Schlüsselwert x (mit Stopper-Technik)

```
public boolean search(int x) {
  KettenElement element = head.next;
  tail.key = x;
  while (element.key != x)
    element = element.next;
  return (element != tail);
}
```

Verketten von zwei Listen

- Aneinanderfügen von aktueller Liste und L ergibt neue Liste

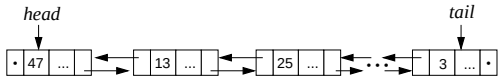
```
public KettenListe2 concat (KettenListe2 L) {
  KettenListe2 liste = new KettenListe2();
  liste.head = head;
  tail.next.next = L.head.next;
  liste.tail = L.tail;
  if (L.tail.next == L.head) // leere Liste L
    liste.tail.next = tail.next;
  return liste;
}
```


Verkettete Liste: Implementierung 3

Implementierung 3: doppelt gekettete Liste

```
class KettenElement2 {
  int key;
  String wert;
  KettenElement2 next = null;
  KettenElement2 prev = null;
}

// Liste
class KettenListe3 implements Liste {
  KettenElement2 head = null;
  KettenElement2 tail = null;
  /** Konstruktor */
  public KettenListe3() {
    head = new KettenElement2();
    tail = new KettenElement2();
    head.next = tail;
    tail.prev = head;
  }
  ...
}
```



Implementierung 3 (Forts.)

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)

Verkettete Listen

Suchaufwand bei ungeordneter Liste

- erfolgreiche Suche: $C_{avg} = \frac{n+1}{2}$ (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
- lineare Kosten für Einfügen in Sortierreihenfolge

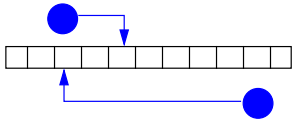
VERGLEICH der 3 Implementierungen	Implem. 1	Implem. 2	Implem. 3 (Doppelkette)
Einfügen am Listenanfang			
Einfügen an gegebener Position			
Löschen an gegebener Position			
Suchen eines Wertes			
Hintereinanderfügen von 2 Listen			

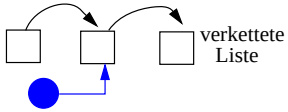
Iterator-Konzept

Problem: Navigation der Listen ist implementationsabhängig

Iterator-Konzept ermöglicht einheitliches sequentielles Navigieren

- Iterator ist Objekt zum Iterieren über Kollektionen (Listen, Mengen ...)
- mehrere Iteratoren auf einer Kollektion möglich





Java-Schnittstelle für Iteratoren `java.util.Iterator` mit folgenden Methoden:

- `boolean hasNext()`
  liefert true wenn weitere Elemente in Kollektion verfügbar, ansonsten false
- `Object next()`
  liefert das aktuelle Element und positioniert auf das nächste Element
- `void remove()`
  löscht das aktuelle Element

Iterator-Konzept (2)

Implementierung am Beispiel der einfach verketteten Liste

```
class KettenListe1 implements Liste {
  class ListIterator implements java.util.Iterator {
    private KettenElement element = null;
    /** Konstruktor */
    public ListIterator(KettenElement e) { element = e; }

    public boolean hasNext() { return element != null; }

    public void remove() {
      throw new UnsupportedOperationException(); }

    public Object next() {
      if(!hasNext())throw new java.util.NoSuchElementException();
      Object o = element.wert;
      element = element.next;
      return o; }
  }
  public java.util.Iterator iterator() {
    return new ListIterator(head); }
  ...
}
```


Iterator-Konzept (3)

Verwendung von Iteratoren

```
KettenListe1 liste = new KettenListe1();
...

java.util.Iterator iter = liste.iterator();
String wert = null;

while (iter.hasNext()) {
  wert = (String) iter.next();
  ...
}
```


Häufigkeitsgeordnete lineare Listen

sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind

mittlere Suchkosten: $C_{avg}(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + n \cdot p_n$
für Zugriffswahrscheinlichkeiten p_i

Zur Minimierung der Suchkosten sollte Liste direkt so aufgebaut oder umorganisiert werden, daß $p_1 \geq p_2 \geq \ldots \geq p_n$

Beispiel: Zugriffsverteilung nach 80-20-Regel

- 80% der Suchanfragen betreffen 20% des Datenbestandes und von diesen 80% wiederum 80% (also insgesamt 64%) der Suchanfragen richten sich an 20% von 20% (insgesamt 4%) der Daten.
- Erwarteter Suchaufwand $C_{avg}(n) =$

Da Zugriffshäufigkeiten meist vorab nicht bekannt sind, werden selbstorganisierende (adaptive) Listen benötigt

Selbstorganisierende Listen

Ansatz 1: FC-Regel (Frequency count)

- Führen von Häufigkeitszählern pro Element
- Jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1
- falls erforderlich, wird danach die Liste lokal neu geordnet, so daß die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
- hoher Wartungsaufwand und Speicherplatzbedarf

Ansatz 2: T-Regel (Transpose)

- das Zielelement eines Suchvorgangs wird dabei mit dem unmittelbar vorangehenden Element vertauscht
- häufig referenzierte Elemente wandern (langsam) an den Listenanfang

Ansatz 3: MF-Regel (Move-to-Front)

- Zielelement eines Suchvorgangs wird nach jedem Zugriff 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)

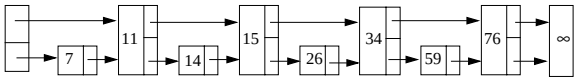
Skip-Listen

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

- Verwendung sortierter verkettet gespeicheter 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



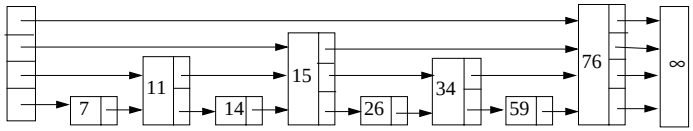
Suche:

- beginnt auf oberster Ebene bis Element E gefunden wird, dessen Schlüssel den Suchschlüssel übersteigt (dabei werden viele Elemente übersprungen)
- Fortsetzung der Suche auf darunterliegender Ebene bei Elementen, die nach dem Vorgänger von E folgen
- Fortsetzung des Prozesses bis auf Ebene 0

Skip-Listen (2)

Perfekte Skip-Liste:

- Anzahl der Ebenen (Listenhöhe): 1 + log n



- max. Gesamtanzahl der Zeiger:
- Suche: O (log n)

Perfekte Skip-Listen zu aufwendig bezüglich Einfügungen und Löschvorgängen

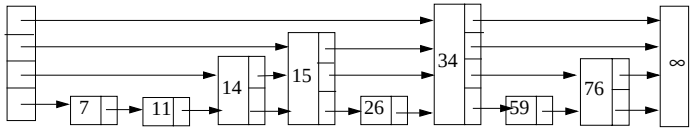
- vollständige Reorganistion erforderlich
- Kosten O (n)

Skip-Listen (3)

Abhilfe: Randomisierte Skip-Listen

- strikte Zuordnung eines Elementes zu einer Ebene ("Höhe") 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 (Höhe von x = i) = $1 / 2^i$ (für alle i)
- 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

Synonyme: Stapel, Keller, LIFO-Liste usw.

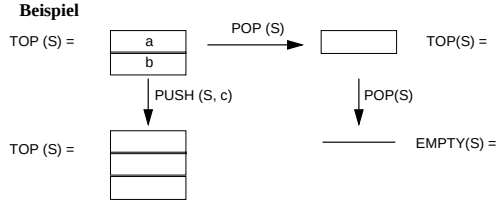
Stack kann als spezielle Liste aufgefaßt werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt, vorgenommen werden

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)

Beispiel



Stacks (2)

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: $s_0 \in$ STACK
- Stack-Operationen werden durch ihre Funktionalität charakterisiert. Ihre Semantik wird durch Axiome festgelegt.

Definitionen:

Datentyp STACK
Basistyp ELEM

Operationen:

CREATE:		→	STACK;
INIT:	STACK	→	STACK;
PUSH:	STACK × ELEM	→	STACK;
POP:	STACK - {s_0}	→	STACK;
TOP:	STACK - {s_0}	→	ELEM;
EMPTY:	STACK	→	{TRUE, FALSE}.

Axiome:

CREATE = s_0;
EMPTY (CREATE) = TRUE;

∀s ∈ STACK, ∀x ∈ ELEM:

INIT (s) = s_0;
EMPTY (PUSH(s , x)) = FALSE;
TOP (PUSH (s , x)) = x;
POP (PUSH (s , x)) = s;
NOT EMPTY (s) ⇒ PUSH (POP(s) , TOP(s)) = s

Stacks (3)

Interface-Definition

```
public interface Stack {
  public void push(Object o) throws StackException;
  public Object pop() throws StackException;
  public Object top() thrwos StachException;
  public boolean isempty();
}
```

Array-Implementierung des Stack-Interface

```
public class ArrayStack implements Stack {
  private Object elements[] = null;
  private int count = 0;
  private final int defaultSize = 100;

  public ArrayStack(int size) {
    elements = new Object[size];
  }

  public ArrayStack() {
    elements = new Object[defaultSize];
  }
```


```
public void push(Object o) throws StackException {
  if(count == elements.length-1)
    throw new StackException("Stack voll!");
  elements[count++] = o;
}

public Object pop() throws StackException {
  if(isempty())
    throw new StackException("Stack leer!");
  Object o = elements[--count];
  elements[count] = null; // Freigeben des Objektes
  return o;
}

public Object top() throws StackException {
  if(isempty())
    throw new StackException("Stack leer!");
  return elements[count-1];
}

public boolean isempty() {
  return count == 0;
}
```


Stacks (4)

Anwendungsbeispiel 1: Erkennen wohlgeformter Klammerausdrücke

Definition

- () ist ein wohlgeformter Klammerausdruck (wgK)
- Sind w1 und w2 wgK, so ist auch ihre Konkatenation w1 w2 ein wgK
- Mit w ist auch (w) ein wgK
- Nur die nach den vorstehenden Regeln gebildeten Zeichenreihen bilden wgK

Lösungsansatz

- Speichern der öffnenden Klammern in Stack
- Entfernen des obersten Stack-Elementes bei jeder schließenden Klammer
- wgK liegt vor, wenn Stack am Ende leer ist

Realisierung

```
public boolean wgK(String ausdruck) {
  Stack stack = new ArrayStack();
  char ch;
  for (int pos=0; pos < ausdruck.length(); pos++) {
    ch = ausdruck.charAt(pos);
    if (ch == '(') stack.push(new Character(ch));
    else if (ch == ')') {
      if (stack.isempty()) return false;
      else stack.pop();
    }
  }
  if (stack.isempty()) return true;
  else return false;
}
```


Stacks (5)

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

Beispiel: (a+b) x (c+d/e) => a b + c d e / + x

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:

UPN	a	b	+	c	d	e	/	+	x

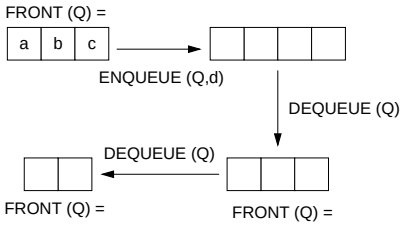
Platz 1									
Platz 2									
Platz 3									
Platz 4									

Schlangen

Synonyme: FIFO-Schlange, Warteschlange, Queue

spezielle Liste, bei der die Elemente an einem Ende (hinten) eingefügt und am anderen Ende (vorne) entfernt werden

Beispiel:



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

Schlangen (2)

formale ADT-Spezifikation

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

Datentyp QUEUE
Basistyp ELEM

Operationen:

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

Axiome:

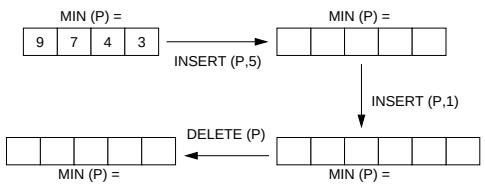
CREATE= q_0;
EMPTY (CREATE) = TRUE;

$\forall q \in$ QUEUE, $\forall x \in$ ELEM:
INIT (q) = q_0;
EMPTY (ENQUEUE(q, x)) = FALSE;
EMPTY (q) $\Rightarrow$ FRONT (ENQUEUE(q , x)) = x;
EMPTY (q) $\Rightarrow$ DEQUEUE (ENQUEUE(q , x) = q;
NOT EMPTY (q) $\Rightarrow$ FRONT (ENQUEUE(q , x)) = FRONT(q);
NOT EMPTY (q) $\Rightarrow$ DEQUEUE (ENQUEUE(q , x)) = ENQUEUE(DEQUEUE(q), x).

Vorrangwarteschlangen

Vorrangwarteschlange (priority queue)

- 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

Vorrangwarteschlangen (2)

formale ADT-Spezifikation :

Datentyp PQUEUE
Basistyp ELEM
(besitzt totale Ordnung $\leq$)

leere Vorrangwarteschlange:
$p_0 \in$ PQUEUE

Operationen:

CREATE:		$\rightarrow$	PQUEUE;
INIT:	PQUEUE	$\rightarrow$	PQUEUE;
INSERT:	PQUEUE $\times$ ELEM	$\rightarrow$	PQUEUE;
DELETE:	PQUEUE - $\{p_0\}$	$\rightarrow$	PQUEUE;
MIN:	PQUEUE - $\{p_0\}$	$\rightarrow$	ELEM;
EMPTY:	PQUEUE	$\rightarrow$	{TRUE, FALSE}.

Axiome:

CREATE = p_0;
EMPTY (CREATE) = TRUE;

$\forall p \in$ PQUEUE, $\forall x \in$ ELEM:
INIT (p) = p_0;
EMPTY (INSERT (p, x)) = FALSE;
EMPTY (p) $\Rightarrow$ MIN (INSERT (p, x)) = x;
EMPTY (p) $\Rightarrow$ DELETE (INSERT (p, x)) = p;
NOT EMPTY (p) $\Rightarrow$ IF x $\leq$ MIN (p) THEN MIN (INSERT (p, x)) = x
ELSE MIN (INSERT (p, x)) = MIN(p);
NOT EMPTY (p) $\Rightarrow$ IF x $\leq$ MIN (p) THEN DELETE (INSERT (p , x)) = p
ELSE DELETE (INSERT (p, x)) = INSERT (DELETE (p), x);

Zusammenfassung

Verkettete Listen

- dynamische Datenstrukturen mit geringem Speicheraufwand und geringem Änderungsaufwand
- Implementierungen: einfach vs. doppelt verkettete Listen
- hohe Flexibilität
- hohe Suchkosten

Iterator-Konzept: implementierungsunabhängige Navigation in Kollektionen (u.a. Listen)

Adaptive (selbstorganisierende) Listen erlauben reduzierte Suchkosten

- Nutzung von Lokalität bzw. ungleichmäßigen Zugriffshäufigkeiten
- Umsetzung z.B. über Move-to-Front oder Transpose

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