Inhalt

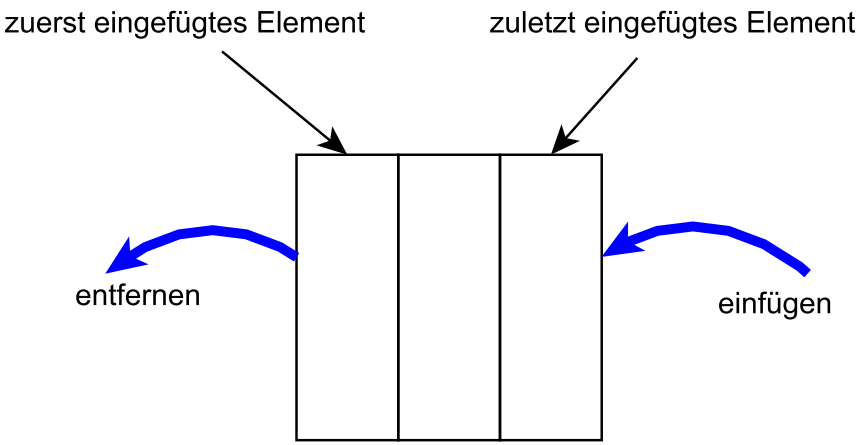
FIFO

Andere häufige Anforderung an eine Sammlung von Datensätzen:

- Datensätze sollen hinzugefügt und entfernt werden können.
- Die Reihenfolge der Einfügungen ist für das Entfernen von Bedeutung:
- Es kann immer nur das zuerst eingefügte Element wieder entfernt werden.

Diese Strategie ist unter dem Namen „FIFO“ (Abk. für first in - first out) bekannt.

Sie entspricht der typischen Warteschlange.



ADT (Warte-)Schlange

Wir definieren einen abstrakten Datentyp Schlange durch die Operationen

- T front(): liefert das „vorderste“ Element der Schlange
- **void** enqueue(T): fügt ein neues Element in die Schlange ein
- **void** dequeue(): entfernt das vorderste Element

- Wie bei Stapeln sind Duplikate erlaubt
- Unterschied zum ADT Stapel:
  Einfügen und Löschen wirken auf die beiden verschiedenen Enden der Schlange

Implementierung mittels Arrays

Wir implementieren das Einfügen am „hinteren“ Ende des Arrays und das Entfernen am „Anfang“ des Arrays:
(umgekehrt wäre ebenfalls möglich)

- front() entspricht get(0)
- enqueue(e) entspricht add(e)
- dequeue() entspricht remove()

Analyse:

- front() und enqueue(e) verursachen Kosten von $\mathcal{O}(1)$
  (Analyse von enqueue() bzw add(e) wie bei Stapeln)
- dequeue() verursacht Kosten von $\mathcal{O}(n)$
  (durch das notwendige „Nachrücken“ aller Folgeelemente)

Bei umgekehrter Implementierung (Einfügen vorne, Entfernen am hinteren Ende des Arrays) würde sich ergeben, dass enqueue() Kosten von $\mathcal{O}(n)$ hat, während dann dequeue() in konstanter Zeit läuft.

Implementierung mittels Verketteter Liste

Bei Verwendung einer EVL mit last-Referenz gilt:

- front() entspricht getFirst()
- enqueue(e) entspricht append(e)
- dequeue() entspricht removeFirst()

Analyse:

- Bei keiner der Operationen muss in einer Schleife die Liste ganz oder teilweise durchlaufen werden $\leadsto$
- alle drei Operationen benötigen nur konstanten Zeitaufwand: $\mathcal{O}(1)$

Vergleich der Implementierungen

- Auf den ersten (theoretischen) Blick scheint eine Implementierung mittels EVL günstiger (alle Operationen in konstanter Zeit) als eine Implementierung mittels Arrays (eine der Operation nur in linearer Zeit)
- ... insbesondere bei Anwendungen mit großer „Fluktuation“ des Datenbestands (viele enqueue()- und dequeue()-Operationen).
- In der Praxis ist aber der Aufwand für das Erzeugen neuer Listenelemente (was in der theoretischen Überlegung als Elementaroperation gezählt wird) so hoch, dass in Real-Zeit-Messungen die Implementierung mittels Arrays besser ist.

Inhalt

Datenstruktur Ringpuffer

Ein Ringpuffer ist eine array-basierte Datenstruktur, mit der sehr effizient die Operationen `addLast(T e)` und `removeFirst()` implementiert werden können.

Voraussetzung

- die maximal vorkommende Größe des Datenbestands ist bekannt
- oder: man interessiert sich nur für eine bekannte feste maximale Anzahl an Datensätzen

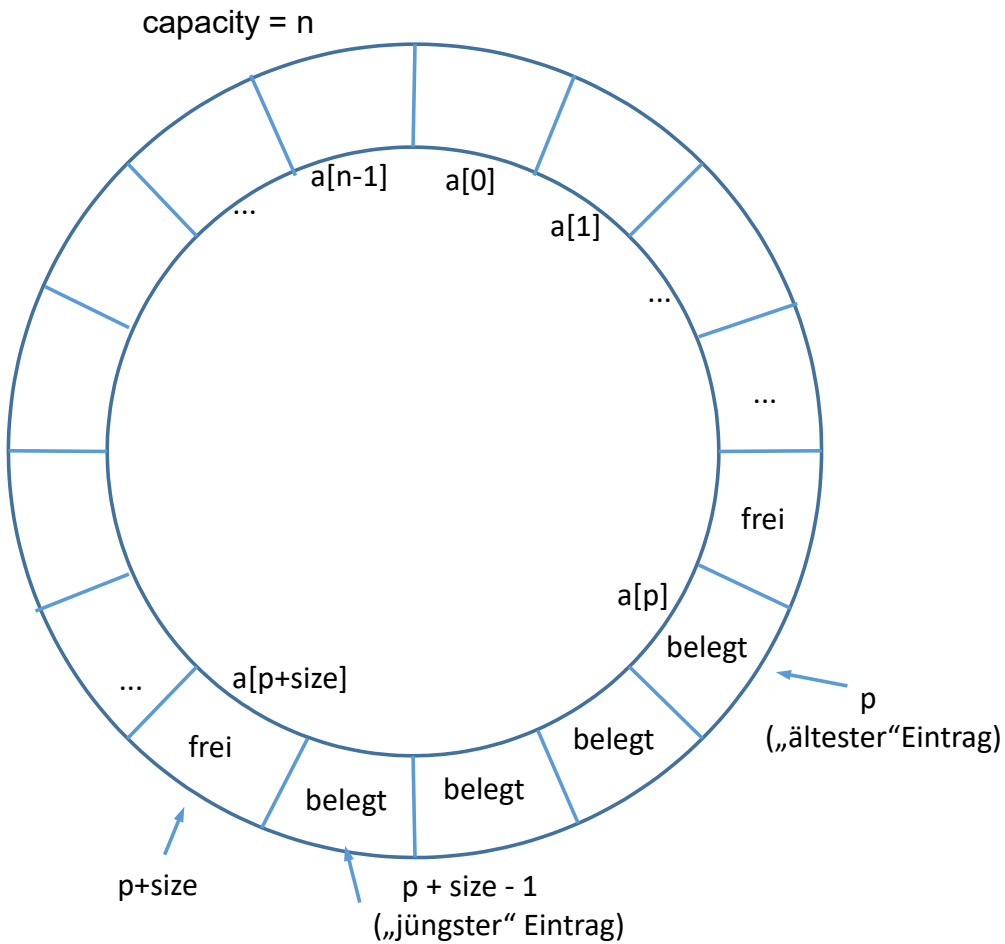
Idee:

- Speichere die Daten in einem Array fester Länge.
- Betrachte das Array als geschlossenen Ring,
- der zyklisch durchlaufen wird.

Unterscheide:

- `size`: Anzahl der aktuell gespeicherten Werte
- `capacity`: Anzahl der maximal möglichen gespeicherten Werte

Ringpuffer graphisch



Gute Vorstellung: eine „im Kreis kriechende Schlange“

Datenstruktur Ringpuffer

Es sollen folgende Methoden implementiert werden:

```
int size()
boolean isEmpty()
int capacity()
boolean contains(T e)
void add(T e) // fuegt einen neuen "juengsten" Eintrag ein
void remove() // loescht den "aeltesten" Eintrag
T get() // liefert eine Eintrag, ohne zu loeschen (s.u.)
```

Außerdem sollen Ringpuffer grundsätzlich iterierbar sein.

Die Semantik der beiden Methoden add() und get() betrachten wir in zwei verschiedenen Varianten;
die Wirkung des Iterator soll sich ebenfalls unterscheiden:

Ringpuffer: 2 Varianten

- Ringpuffer „ohne Überschreiben“
  - add() löst eine Exception aus und fügt nichts ein, falls der Puffer voll ist
  - get() liefert den ältesten gepufferten Wert
  - der Iterator liefert die Elemente in der Reihenfolge „ältester“, „zweitältester“ ...
- Ringpuffer „mit Überschreiben“
  - add() überschreibt den ältesten Eintrag, falls der Puffer voll ist
  - get() liefert den jüngsten gepufferten Wert
  - der Iterator liefert die Elemente in der Reihenfolge „jüngster“, „zweitjüngster“ ...

Implementierung Ringpuffer

- nutze einen oder zwei int-„Zeiger“ für die nächste zu belegende Array-Position bzw. für die Position des ältestens Eintrags
- berechne daraus (und ggf aus size) die Indizes (für get(), add() und remove()) jeweils modulo der Array-Länge n

Analyse:

- Alle Methoden benötigen nur konstanten Zeitaufwand: $\mathcal{O}(1)$
- Der Platzbedarf ist ebenfalls konstant (beschränkt durch capacity): $\mathcal{O}(1)$

Beispiel-Implementierung für einen Ringpffer ohne Überschreiben:
Ringpuffer.java und RingpufferFIFO.java

Beispiel-Verlauf für einen RingpufferFIFO

Für einen Integer-RingpufferFIFO mit capacity = 4 und der angegebenen Anweisungsfolge stellt die folgende Speichertabelle den Verlauf dar:

Operation	p	size	Array-Inhalt a[0]	a[1]	a[2]	a[3]	get() - returnWert
	0	0					
add(18); get()	0	1	**18**	-	-	-	18
add(72); get()	0	2	**18**	**72**	-	-	18
add(35); get()	0	3	**18**	**72**	**35**	-	18
remove(); get()	1	2	18	**72**	**35**	-	72
remove(); get()	2	1	18	72	**35**	-	35
add(41); get()	2	2	18	72	**35**	**41**	35
add(25); get()	2	3	**25**	72	**35**	**41**	35
add(42); get()	2	4	**25**	**42**	**35**	**41**	35
remove(); get()	3	3	**25**	**42**	35	**41**	41