5. Listen

- Wichtige abstrakte Datentypen in Zusammenhang mit Listen kennen,
- Implementierungsansätze für Warteschlangen verstehen, sowie deren Vor- und Nachteile benennen können,
- den Aufbau von verketteten Listen und die zugehörigen Algorithmen implementieren können,
- die Effizienz von Listenoperationen einschätzen können und
- den Begriff der amortisierten Laufzeitanalyse kennen und einordnen können.

Warteschlange

Definition 5.1. Es sei T ein beliebiger Datentyp. Eine *Warteschlange (Queue)* ist eine Datenstruktur, die die folgenden Operationen unterstützt:

- `void enqueue(T elem)`
  Fügt ein Element `elem` an das Ende der Warteschlange an.
- `void dequeue()`
  Entfernt das erste Element der Warteschlange.
- `T front()`
  Liefert das Erste Element der Warteschlange, ohne es zu entfernen.

Beispiel: Warteschlange

enqueue(4711); enqueue(7643); enqueue(1234);

4711	7643	1234

dequeue(); enqueue(1999); enqueue(2812); enqueue(32168);

7643	1234	1999	2812	32168

dequeue(); dequeue(); dequeue(); enqueue(1683);

2812	32168	1683

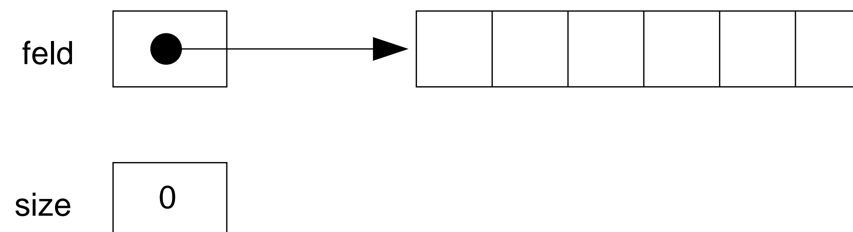
Bemerkungen: Warteschlange

- Die Warteschlange arbeitet nach dem *FIFO-Prinzip (first in, first out)*.
- Im Gegensatz dazu ein Stack: LIFO (last in, first out)
- Die generische Schnittstelle `Queue` (aus dem Paket `java.util`) entspricht der Spezifikation einer Warteschlange.
- Es gibt in `java.util` verschiedene Klassen, die die Schnittstelle `Queue` implementieren.
- Wir interessieren uns für Implementierungsansätze und deren Eigenschaften (Effizienz der Operationen).

Implementierungsansatz: Feld

- Wir speichern die Elemente der Warteschlange in einem Feld fester Größe. In einer weiteren Variablen (`size`) merken wir uns, wie viele Feldelemente belegt sind.

```
public class Warteschlange<T> {
    private final int FIXED_MAX_SIZE = ...;
    ...
    private T[] feld = (T[]) new Object[FIXED_MAX_SIZE];
    private int size = 0;
    ...
}
```



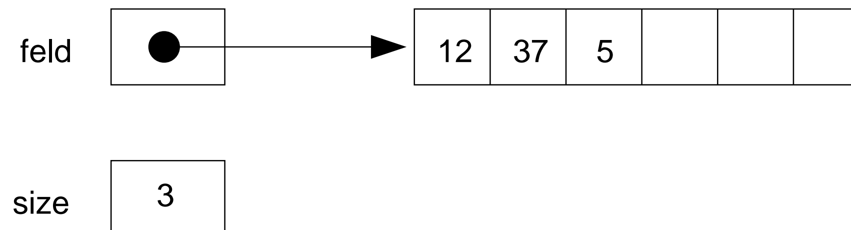
- `void enqueue(T elem)`

  Wir legen `elem` an der ersten freien Stelle im Feld ab und inkrementieren `size`.

  ```
  feld[size] = elem;
  size++;
  ```

  Dies geht natürlich nur, wenn die Kapazität des Feldes noch nicht erschöpft ist, d.h. `size < FIXED_MAX_SIZE` gilt.

  Beispiel: `enqueue(12); enqueue(37); enqueue(5);`



- T front()

  Im Normalfall (`size > 0`) geben wir einfach das erste Element des Feldes zurück:

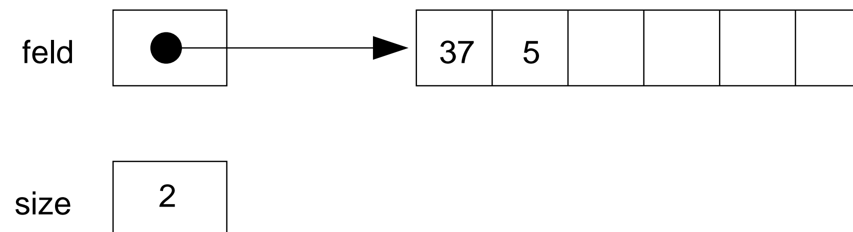
  ```
  return feld[0];
  ```

- `void dequeue()`

  Wir verschieben alle Elemente ab der zweiten Position im Feld um eine Position nach vorne und dekrementieren `size`.

```
for(int i=1 ; i<size ; i++) {
   feld[i-1] = feld[i];
}
size--;
```

Beispiel: `dequeue();`

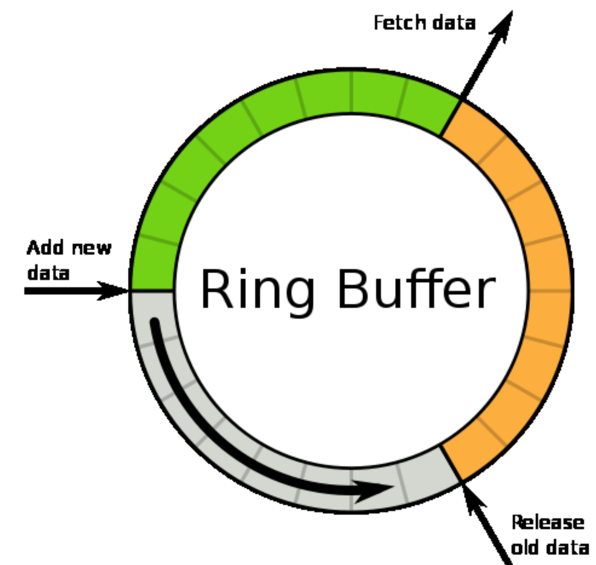


Effizienz

- Es sei n die Anzahl der Elemente in der Warteschlange.
- Zeitaufwand für `enqueue()` und `front()`: $O(1)$
- Zeitaufwand für `dequeue()`: $O(n)$
- Speicherplatzverbrauch: $O(n)$
- Probleme:
  - Bei `size == FIXED_MAX_SIZE` ist kein `enqueue()` mehr möglich.
  - Alternative: Neues größeres Feld anlegen und Inhalte kopieren, dann aber Zeitaufwand nicht mehr $O(1)$ sondern $O(n)$ für `enqueue()`.

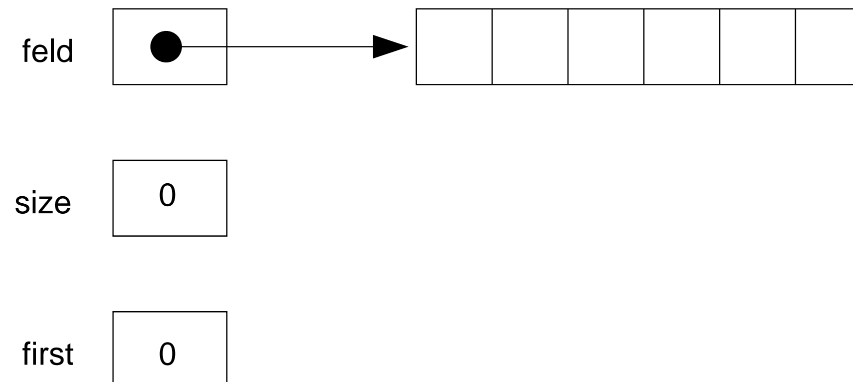
Implementierungsansatz: Ringpuffer

- Bisher: Hoher Zeitaufwand durch das Verschieben der Elemente bei `dequeue()`.
- Besser: Statt die Elemente zu verschieben verschiebt man den Anfangsindex. Diesen merkt man sich in einer weiteren Instanzvariablen.
- Entspricht logisch einem sogenannten *Ring-Puffer*.



- In einer zusätzlichen Variablen (`first`) merken wir uns, bei welcher Feldposition die Warteschlange beginnt.

```
public class Warteschlange<T> {
    public final int FIXE_MAX_SIZE = ...;
    ...
    private T[] feld = (T[]) new Object[FIXED_MAX_SIZE];
    private int size = 0;
    private int first = 0;
    ...
}
```



- `void enqueue(T elem)`

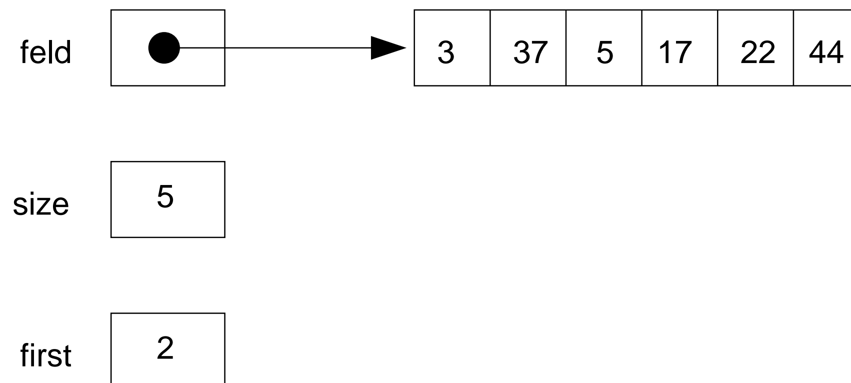
  Man beachte, dass die Warteschlange nicht unbedingt bei Index 0 beginnt! Bei Überlauf wird wieder ab vorne wieder aufgefüllt.

  ```
  feld[(first+size) % FIXED_MAX_SIZE] = elem;
  size++;
  ```

  Dies geht wiederum nur, wenn die Kapazität des Feldes noch nicht erschöpft ist.

  Beispiel:

  ```
  enqueue(12); enqueue(37); enqueue(5); dequeue(); dequeue(); enqueue(17);
  enqueue(22); enqueue(44); enqueue(3);
  ```



Erstes Element der Warteschlange bei Index `first`.

Letztes Element der Warteschlange bei Index `(first+size-1)%FIXED_MAX_SIZE`.

Das Element 37 gehört nicht zur Warteschlange!

- T front()

  Falls `size > 0`: Das erste Element der Warteschlange liegt in der Feldkomponente mit Index `first`:

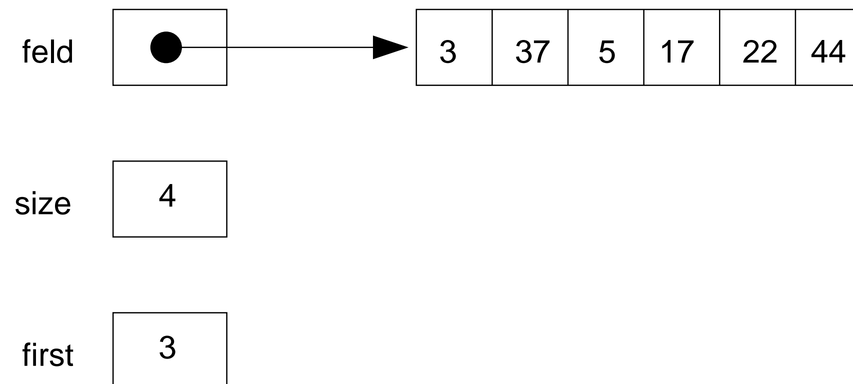
```
return feld[first];
```

- `void dequeue()`

  Wir inkrementieren `first`, müssen dabei aber einen möglichen Überlauf berücksichtigen, und dekrementieren `size`.

```
first = (first+1) % FIXED_MAX_SIZE;
size--;
```

Beispiel: `dequeue();`



Beginn der Warteschlange jetzt bei Element 17, Ende bei 3, 37 und 5 gehören nicht dazu.

Effizienz

- Zeitaufwand für alle Operationen: $O(1)$
- Es bleibt das Problem der beschränkten Kapazität, damit liegt keine dynamische Datenstruktur vor.

  Als *dynamische Datenstruktur* bezeichnet man Datenstrukturen, die eine flexible Menge an Arbeitsspeicher reservieren.
- Die Erzeugung eines neuen größeren Feldes führt zu einem Zeitaufwand von $O(n)$ für `enqueue()`.

Deque

Eine *Deque (Double-Ended Queue)* erlaubt das Einfügen, Löschen und den Zugriff sowohl am Anfang als auch am Ende der Elementliste.

- `void insert(T elem)`
  Fügt ein Element `elem` am Anfang der Elementliste ein.
- `void append(T elem)`
  Fügt ein Element `elem` am Ende der Elementliste ein (entspricht `enqueue()`).
- `void deleteFront()`
  Entfernt das erste Element der Elementliste (entspricht `dequeue()`).
- `void deleteRear()`
  Entfernt das letzte Element der Elementliste.
- `T front()`
  Liefert das erste Element.

- `T rear()`
  Liefert das letzte Element.

Prinzipiell sind die bisher gezeigten Implementierungsansätze auch für eine Deque geeignet, mit den schon vorgestellten Vor- und Nachteilen.