

## 3. Lineare Zugriffe und co

Quelle: [ep2-03\\_lineare-Zugriffe\\_Assoziative-Zugriffe\\_Implementierungsdetails.pdf](#)

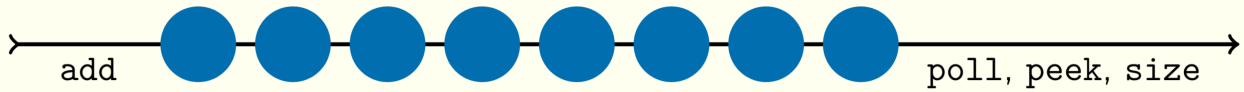
Beinhaltet: Lineare Zugriffe, Assoziative Zugriffe, Implementierungsdetails

---

### Lineare Zugriffe auf Daten

- **Prinzip des linearen Zugriffs:**
    - **Ein Element wird hineingegeben**, die Daten werden in eine Sammlung aufgenommen.
    - **Nächstes Element wird herausgeholt**, und so weiter.
    - **Kein aufwendiges Adressieren oder Indexieren** erforderlich.
  - **Eigenschaften:**
    - **Einfache Verwendung:**
      - Daten können sequentiell durchlaufen und bearbeitet werden, ohne komplexe Operationen oder Berechnungen.
    - **Häufig keine Größenfestlegung nötig:**
      - Bei vielen linearen Datenstrukturen muss die Größe nicht im Voraus definiert werden, was eine flexible Handhabung der Daten erlaubt.
-

## Queue



FIFO-Verhalten (first-in, first-out)

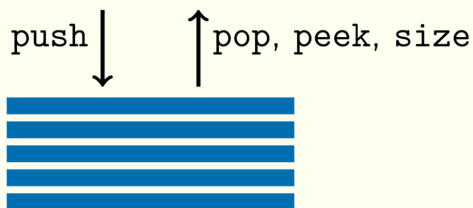
**add** Einfügen sollte in der Regel funktionieren (mitwachsende Datensammlung)

**poll** Lesen und Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden

**peek** Lesen ohne Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden

**size** Anzahl der Einträge, nötig wenn `null` als Eintrag erlaubt

## Stack



LIFO-Verhalten (last-in, first-out)

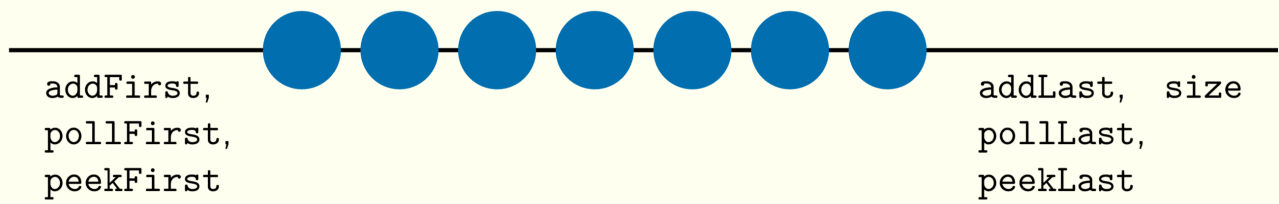
**push** Einfügen sollte in der Regel funktionieren (mitwachsende Datensammlung)

**pop** Lesen und Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden

**peek** Lesen ohne Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden

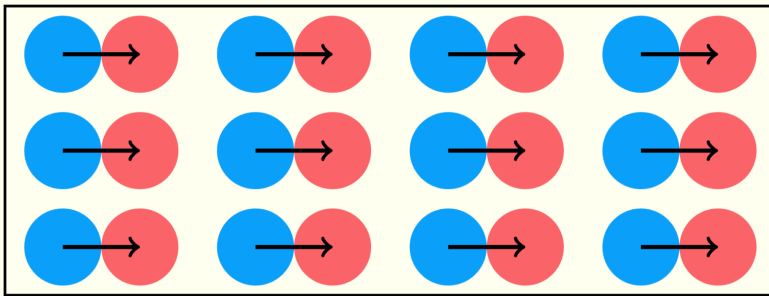
**size** Anzahl der Einträge, nötig wenn `null` als Eintrag erlaubt

## Double-Ended-Queue



symmetrische Zugriffe auf beiden Seiten → Verhalten variabel (FIFO, LIFO, ...)

# Assoziative Datenstruktur



```

put( key , value )
get( key )
remove( key )
containsKey( key )
containsValue( value )
size()

```

beliebig viele unterschiedliche **Schlüssel** mit je einem **Wert** assoziiert

**Eintrag** ist Kombination aus Schlüssel und assoziiertem Wert

wahlfreier Zugriff auf Werte über Schlüssel (beliebig oft zugreifbar)

Datensammlung wächst meist mit der Anzahl der Einträge

## Methoden einer AD

<code>put(k, v)</code>	assoziiert Schlüssel <code>k</code> mit neuem Wert <code>v</code> , gibt alten Wert zurück (oder <code>null</code> wenn Eintrag neu)
<code>get(k)</code>	gibt mit Schlüssel <code>k</code> assoziierten Wert zurück (oder <code>null</code> wenn <code>k</code> mit keinem Wert assoziiert ist)
<code>remove(k)</code>	entfernt Eintrag mit Schlüssel <code>k</code> (falls er existiert), gibt vorher mit <code>k</code> assoziierten Wert zurück (oder <code>null</code> )
<code>containsKey(k)</code>	<code>true</code> wenn ein Eintrag mit Schlüssel <code>k</code> existiert
<code>containsValue(v)</code>	<code>true</code> wenn es einen Eintrag mit Wert <code>v</code> gibt
<code>size()</code>	Anzahl der Einträge

--> [1. Einführung > Arten von Methoden](#)

## Assoziative Datenstruktur vs. Array

### Array

- **Indexierung:**
  - Der **Index** im Array ist eine **ganze Zahl** in einem **fortlaufenden Indexbereich**.
- **Größe:**

- Die **Größe** des Arrays muss bereits **beim Anlegen bekannt sein**.
- **Indexbereich:**
  - Der **Indexbereich** des Arrays bleibt **über die gesamte Lebensdauer des Arrays konstant**.
- **Zugriffseffizienz:**
  - **Zugriff auf das Array** ist sehr effizient, da die Elemente durch ihre Position im Speicher direkt adressierbar sind.

## Assoziative Datenstruktur

- **Indexierung:**
  - Der **Schlüssel** in einer assoziativen Datenstruktur kann **beliebigen Typ** haben, nicht zwingend eine fortlaufende Zahl.
  - **Kein fortlaufender Bereich** nötig, der Schlüssel kann beliebig gewählt werden (z. B. Strings, Objekte).
- **Größe:**
  - Eine assoziative Datenstruktur **kann nach Bedarf mitwachsen** und muss nicht vorab eine feste Größe haben.
- **Flexibilität:**
  - Einträge in einer assoziativen Datenstruktur (Schlüssel + Wert) sind **hinzufügbar und entfernbar**. Sie können dynamisch angepasst werden.
- **Zugriffseffizienz:**
  - Der Zugriff auf eine **assoziative Datenstruktur** ist im Allgemeinen weniger effizient als der Zugriff auf ein Array, da intern oft eine Hash-Tabelle oder ein ähnliches Verfahren verwendet wird, um die Zuordnung zwischen Schlüssel und Wert zu finden.

## Vorteile der assoziativen Datenstruktur

- **Einfache Handhabung:**
    - Der Umgang mit assoziativen Datenstrukturen ist oft einfacher und flexibler, da man nicht mit festen Indexen oder Größen arbeiten muss.
  - **Dynamische Anpassung:**
    - Sie wachsen und schrumpfen nach Bedarf und bieten mehr **Freiheit** bei der Handhabung der Daten.
-

# Ziele bei Implementierung

## Korrektheit:

Implementierung entspricht vorgegebener Außensicht

## Einfachheit ( $\approx$ Wartbarkeit):

so wenige Fallunterscheidungen und Schleifen wie möglich

mehrfache Vorkommen gleicher und ähnlicher Programmtexte vermeiden

nur leicht nachvollziehbare Annahmen treffen

schwer verständliche Textteile vermeiden

## Effizienz:

effiziente Programmerstellung (wichtiger Kostenfaktor)

ausreichend effizienter Programmablauf

# Wrapper

## SQueue als Wrapper auf DEQueue

- **Definition:**
  - SQueue ist ein **Wrapper** auf die DEQueue -Klasse.
  - Der Begriff "Wrapper" bezeichnet eine Klasse, die eine andere Klasse **einbettet** und deren Funktionalität weiterverwendet.
- **Delegierung:**
  - Die Methoden von SQueue delegieren ihre Aufgaben an das Objekt der Klasse DEQueue .
  - **Delegieren** bedeutet, dass die Methodenaufrufe von SQueue an die entsprechenden Methoden des eingebetteten DEQueue -Objekts weitergegeben werden.
- **Beispiel:**

```
public class SQueue {
    private final DEQueue q = new DEQueue(); // DEQueue wird als internes
    Objekt verwendet

    public void add(String e) {
        q.addLast(e); // Delegierung an DEQueue
    }

    public String poll() {
        return q.pollFirst(); // Delegierung an DEQueue
    }
    // Weitere Methoden könnten folgen
}
```

## Funktionsweise des Wrappers

- **Wrapper erzeugt neue Außensicht:**
  - Der Wrapper ( SQueue ) bietet eine **neue Sicht** auf die bestehende Klasse DEQueue .
  - Dies kann z.B. durch:
    - **Andere Namen oder Parameterreihenfolgen** von Methoden,
    - **Vorgegebene Werte für bestimmte Parameter** (z. B. Standardwerte),
    - **Weglassen von Methoden** oder das **Hinzufügen neuer Methoden** geschehen.
  - In diesem Beispiel delegiert SQueue die Aufgaben der Methode add an addLast von DEQueue und die Methode poll an pollFirst .

## Vorteile des Wrappers

- **Anpassung der API:**
  - Mit einem Wrapper kann man die API einer bestehenden Klasse anpassen, ohne die ursprüngliche Klasse zu verändern.
  - Der Wrapper bietet eine **vereinfachte oder angepasste Schnittstelle** für bestimmte Anwendungsfälle.
- **Verstecken von Details:**
  - Details der inneren Implementierung können vor dem Benutzer verborgen werden. Beispielsweise könnte `SQueue` zusätzliche Logik oder Fehlerbehandlung einführen, ohne dass der Benutzer die Details der `DEQueue` kennt.

## Zusammenfassung

- **Delegation** bedeutet, dass ein Objekt die Verantwortung für die Ausführung bestimmter Aufgaben an ein anderes Objekt weitergibt.
  - Ein **Wrapper** bietet eine angepasste Außensicht auf eine bestehende Klasse und verändert die Art und Weise, wie Methoden aufgerufen oder verwendet werden, ohne die ursprüngliche Klasse zu ändern.
-



# Index als Modulo-Wert

## Index als Modulo-Wert

```

public class DEQueue {
    private int mask = (1 << 3) - 1;
    private String[] es = new String[mask + 1];
    private int head, tail;

    public void addFirst(String e) {
        es[head = (head - 1) & mask] = e;
        ...
    }
    public void addLast(String e) {
        es[tail] = e;
        tail = (tail + 1) & mask;
        ...
    }
}

```

Zweierpotenz vorteilhaft, ermöglicht Modulo durch Maskieren

Maske = alle gültigen Bits des Index

Einträge von `es[head]` bis `es[tail-1]` gültig, durch Modulo auch bei `tail < head`, eine Grenze inklusiv, eine exklusiv

zuerst Index dekrementieren (modulo `mask + 1`), dann zugreifen

zuerst zugreifen, dann Index inkrementieren

## Auf null setzen

```

public class DEQueue {
    ...
    public String pollFirst() {
        String result = es[head];
        es[head] = null;
        if (tail != head) {
            head = (head + 1) & mask;
        }
        return result;
    }
    public String peekFirst() {
        return es[head];
    }
}

```

Ergebnis merken

ursprünglichen Eintrag auf `null` setzen  
gut für Programmhygiene, Speicherbereinigung, spart Fallunterscheidungen

`head == tail` wenn ganz leer oder voll, voll nicht möglich weil vorher Array vergrößert, `null` als gültiger Eintrag möglich

`es[head] == null` wenn leer, keine Fallunterscheidung nötig

## Arrays vergrößern

```

public void addFirst(String e) {
    es[head = (head - 1) & mask] = e;
    if (tail == head) { doubleCapacity(); }
}

private void doubleCapacity() {
    mask = (mask << 1) | 1;
    String[] newes = new String[mask + 1];
    int i = 0, j = 0;
    while (i < head) { newes[j++] = es[i++]; }
    j = head += es.length;
    while (i < es.length) { newes[j++] = es[i++]; }
    es = newes;
}

```

sofort verdoppeln wenn voll, Zweierpotenz beibehalten

ein Bit mehr

neues Array

gilt noch immer: tail == head

ab hier: tail != head, Lücke nicht gefüllt

## Vergleich wenn null als Wert zählt

```

public class SimpleAssoc {
    private int top;
    private String[] ks = new String[8];
    private String[] vs = new String[8];

    private int find(String s, String[] a) {
        int i = 0;
        while (i < top && !(s==null ? s==a[i] : s.equals(a[i])))
            i++;
        return i;
    }
    ...
}

```

jeder Array-Index i mit i < top ist gültig

zwei getrennte Arrays für Schlüssel und Werte

kleine Zweierpotenz

Schlüssel oder Werte

Vergleich der Identität für null, Gleichheit sonst wegen komplexem Vergleich zählt sich Methode aus

# Eintragen und Entfernen

## Eintragen – Fallunterscheidungen vermeiden

```
public String put(String k, String v) {
    int i = find(k, ks); i == top → kein Eintrag vorhanden → einfügen
    if (i == top && ++top == ks.length) {
        String[] nks = new String[top << 1];
        String[] nvs = new String[top << 1];
        for (int j = 0; j < i; j++) {
            nks[j] = ks[j]; nvs[j] = vs[j]; Arrays verdoppeln wenn voll, einfaches Umkopieren reicht weil alle Einträge gültig sind
        }
        ks = nks; vs = nvs;
    }
    ks[i] = k; Schlüssel neu eingetragen, egal ob nötig oder nicht
    String old = vs[i];
    vs[i] = v; Wert muss sowieso immer neu eingetragen werden
    return old;
}
```

## Entfernen mit Verschieben eines Eintrags

```
public String remove(String k) {
    int i = find(k, ks);
    String old = vs[i];
    if (i < top) { i == top wenn Schlüssel nicht gefunden
        ks[i] = ks[--top]; Anzahl gültiger Einträge verringern (--top),
        ks[top] = null; Einträge von neuem Index top nach Index i
        vs[i] = vs[top]; (unnötig wenn i == top für neues top),
        vs[top] = null; Einträge an Index top auf null setzen
    }
    return old;
}
```

# Design-Entscheidungen sparen Programmtext

```

public String get(String k) {
    return vs[find(k, ks)];____keine Fallunterscheidung: gefunden/nicht gefunden
}
public boolean containsKey(String k) {
    return find(k, ks) < top;__Gültigkeit eines Eintrags einfach feststellbar
}
public boolean containsValue(String v) {
    return find(v, vs) < top;__gleiche Methode für Suche nach Schlüssel und Wert
}
public int size() {
    return top;_____Anzahl der Einträge entspricht Index in top
}

```