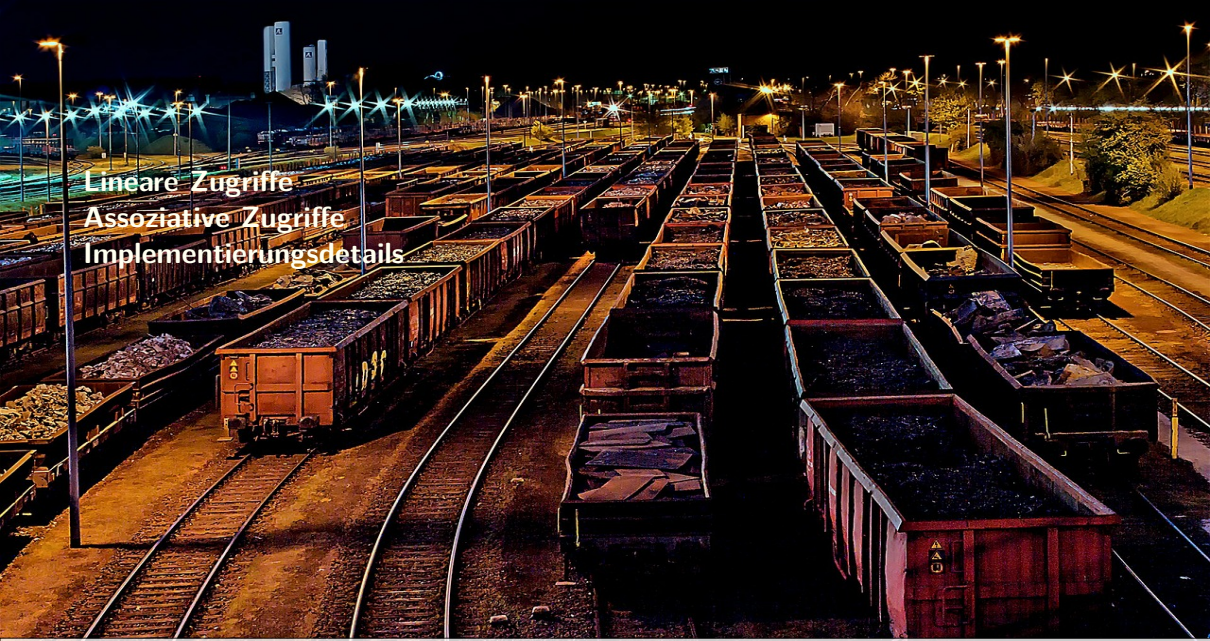

Lineare Zugriffe
Assoziative Zugriffe
Implementierungsdetails

Lineare Zugriffe auf Daten

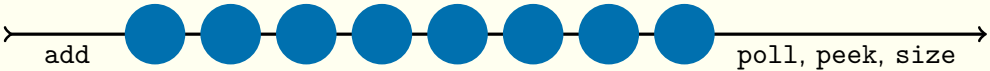
gib ein Element hinein → **Sammlung an Daten** → hole nächstes Element heraus

keine aufwendige Adressierung oder Indexierung

häufig keine Größenfestlegung nötig

einfache Verwendung

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

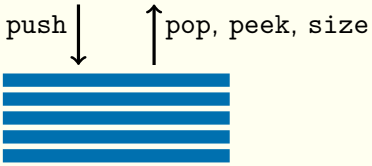
Aufgabe: Wofür eignet sich eine Queue?

In Gruppen zu 2 bis 3 Personen:

Diskutieren Sie, für welche Aufgaben eine Queue gut geeignet ist und für welche nicht.

Zeit: 2 Minuten

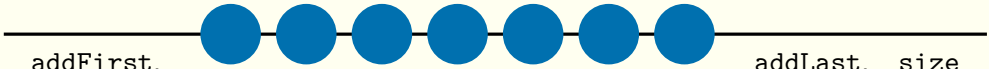
Stack



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



`addFirst`,
`pollFirst`,
`peekFirst`

`addLast`, `size`
`pollLast`,
`peekLast`

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

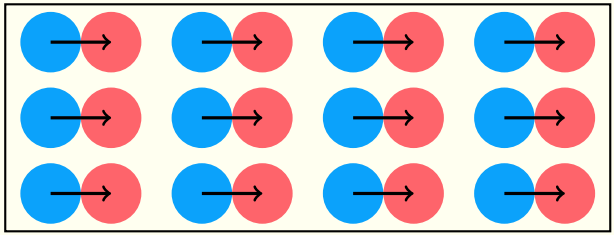
Aufgabe: Wie ist eine Double-Ended-Queue verwendbar?

In Gruppen zu 2 bis 3 Personen:

Welche Kombinationen von `addFirst` und `addLast` mit `pollFirst` und `pollLast` ergeben FIFO-Verhalten, welche LIFO-Verhalten?

Zeit: 2 Minuten

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 assoziativen Datenstruktur

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

Assoziative Datenstruktur im Vergleich zum Array

Array-Index ist ganze Zahl in fortlaufendem Indexbereich,
Schlüssel haben wie Werte beliebigen Typ, kein fortlaufender Bereich nötig

Array-Größe muss beim Anlegen des Arrays bekannt sein,
assoziative Datenstruktur kann meist nach Bedarf mitwachsen

Indexbereich des Arrays bleibt über Lebensdauer des Arrays gleich,
Eingträge (Schlüssel + Wert) in assoziativer Datenstruktur hinzufügbar und entfernbar

Zugriff auf Array ist effizienter als auf assoziative Datenstruktur,
Umgang mit assoziativer Datenstruktur ist einfacher

Aufgabe: Lineare versus assoziative Datenstruktur

Angenommen, Sie müssen in Ihrem Programm eine größere Menge an Daten verwalten.

Anhand welcher Kriterien entscheiden Sie, ob dafür eine lineare oder assoziative Datenstruktur besser geeignet ist?

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

```
public class SQueue {
    private final DEQueue q = new DEQueue();

    public void add(String e) { q.addLast(e); }
    public String poll() { return q.pollFirst(); }
    ...
}
```

SQueue ist **Wrapper** auf DEQueue weil ...

... Methoden ihre Aufgaben an Objekt von DEQueue **delegieren**

Delegieren = Weiterleiten an anderes Objekt

Wrapper erzeugt neue Außensicht auf bestehende Klasse, z.B. durch

- andere Namen oder Parameterreihenfolgen von Methoden,
- vorgegebene Werte für bestimmte Parameter, Weglassen dieser Parameter
- Weglassen von Methoden oder Hinzufügen weiterer Methoden

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

Aufgabe: Warum Zweierpotenzen?

In Gruppen zu 2 bis 3 Personen:

Warum haben mitwachsende Arrays in der Praxis häufig die Größe von Zweierpotenzen?

Warum wird beim Vergrößern häufig verdoppelt (statt additiv erweitert)?

Zeit: 2 Minuten

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

Array 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 zahlt sich Methode aus

Aufgabe: Eintrag von `null` sinnvoll?

In Gruppen zu 2 bis 3 Personen:

Ist es sinnvoll, `null` in eine Queue oder einen Stack einzutragen?

Ist es sinnvoll, `null` als Schlüssel in eine assoziative Datenstruktur einzutragen?

Ist es sinnvoll, `null` als Wert in eine assoziative Datenstruktur einzutragen?

Zeit: 2 Minuten

Eintragen – Fallunterscheidungen vermeiden

```
public String put(String k, String v) {
    int i = find(k, ks);
    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];
        }
        ks = nks;  vs = nvs;
    }
    ks[i] = k;
    String old = vs[i];
    vs[i] = v;
    return old;
}
```

i == top → kein Eintrag vorhanden → einfügen

Arrays verdoppeln wenn voll, einfaches Umkopieren reicht weil alle Einträge gültig sind

Schlüssel neu eingetragen, egal ob nötig oder nicht

Wert muss sowieso immer neu eingetragen werden

Entfernen mit Verschieben eines Eintrags

```
public String remove(String k) {
    int i = find(k, ks);
    String old = vs[i];
    if (i < top) {
        ks[i] = ks[--top];
        ks[top] = null;
        vs[i] = vs[top];
        vs[top] = null;
    }
    return old;
}
```

`i == top` wenn Schlüssel nicht gefunden

Anzahl gültiger Einträge verringern (`--top`),
Einträge von neuem Index `top` nach Index `i`
(unnötig wenn `i == top` für neues `top`),
Einträge an Index `top` auf `null` setzen

Design-Entscheidungen sparen Programmtext

```
public String get(String k) {
    return vs[find(k, ks)];
}
public boolean containsKey(String k) {
    return find(k, ks) < top;
}
public boolean containsValue(String v) {
    return find(v, vs) < top;
}
public int size() {
    return top;
}
```

keine Fallunterscheidung: gefunden/nicht gefunden

Gültigkeit eines Eintrags einfach feststellbar

gleiche Methode für Suche nach Schlüssel und Wert

Anzahl der Einträge entspricht Index in `top`