

3. Lineare Zugriffe und co

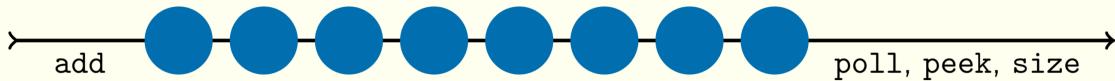
Quelle: ep2-03_lineare-Zugriffe_Arbeitsblatt-Zugriffe_Implementierungsdetails.pdf

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.

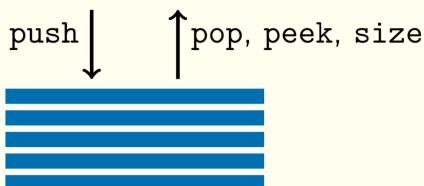
Queue



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

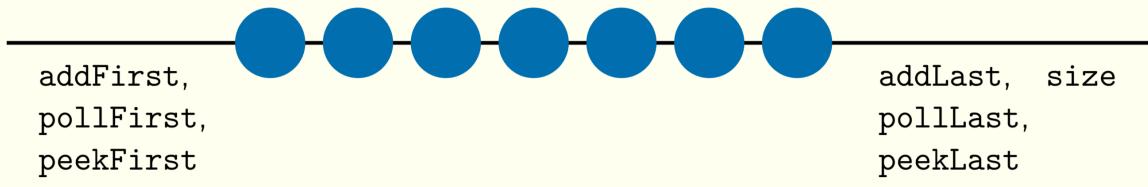
Stack



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

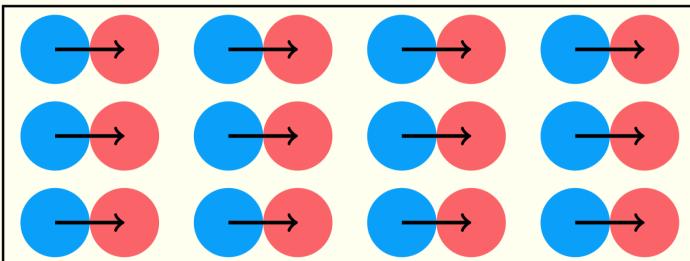
- push** Einfügen sollte in der Regel funktionieren (mitwachsende Datensammlung)
- pop** Lesen und Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- peek** Lesen ohne Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- size** Anzahl der Einträge, nötig wenn `null` als Eintrag erlaubt

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

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

--> [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 entferbar**. Sie können dynamisch angepasst werden.

- **Zugriffseffizienz:**

- Der Zugriff auf eine **assoziativen 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
- mehrfahe 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; Zweierpotenz vorteilhaft,  
ermöglicht Modulo durch Maskieren
    private String[] es = new String[mask + 1];
    private int head, tail; 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
    public void addFirst(String e) {
        es[head = (head - 1) & mask] = e;
        ... zuerst Index dekrementieren (modulo mask + 1), dann zugreifen
    }
    public void addLast(String e) {
        es[tail] = e; zuerst zugreifen, dann Index inkrementieren
        tail = (tail + 1) & mask;
        ...
    }
}

```

Schnelle Bitshift Wiederholung:

Bitwise Shift Operatoren in Java

Bitwise Shift Operatoren manipulieren die einzelnen Bits einer Zahl. In Java gibt es drei Haupttypen:

Arten von Bitwise Shift Operatoren

- **Left Shift (`<<`)**: Verschiebt die Bits nach links. Freie Stellen auf der rechten Seite werden mit Nullen aufgefüllt.
- **Signed Right Shift (`>>`)**: Verschiebt die Bits nach rechts. Das Vorzeichenbit (das höchstwertigste Bit) bleibt erhalten und wird in die frei werdenden Stellen auf der linken Seite kopiert. Dies erhält das Vorzeichen der ursprünglichen Zahl.
- **Unsigned Right Shift (`>>>`)**: Verschiebt die Bits nach rechts. Freie Stellen auf der linken Seite werden immer mit Nullen aufgefüllt, unabhängig vom Vorzeichenbit.

Funktionsweise

Left Shift (`<<`)

Ein Left Shift um `n` Stellen (`x << n`) ist äquivalent zur Multiplikation der Zahl `x` mit 2^n .

Beispiel:

```

int a = 5;      // Binär: 00000101
int b = a << 2; // Binär: 00010100 (entspricht 20)

```

```
System.out.println(b); // Ausgabe: 20
```



Signed Right Shift (`>>`)

Ein Signed Right Shift um `n` Stellen (`x >> n`) ist äquivalent zur Division der Zahl `x` durch 2^n , wobei das Ergebnis auf die nächste ganze Zahl abgerundet wird und das Vorzeichen erhalten bleibt.

Beispiel:

```
int a = 20;      // Binär: 00010100
int b = a >> 2; // Binär: 00000101 (entspricht 5)
System.out.println(b); // Ausgabe: 5

int c = -20;     // Binär (als 32-Bit Zweierkomplement): ...11101100
int d = c >> 2; // Binär: ...11111011 (entspricht -5)
System.out.println(d); // Ausgabe: -5
```

Unsigned Right Shift (`>>>`)

Ein Unsigned Right Shift um `n` Stellen (`x >>> n`) verschiebt die Bits nach rechts und füllt die linken Stellen immer mit Nullen. Das Vorzeichen geht dabei verloren.

Beispiel:

```
int a = -20;      // Binär (als 32-Bit Zweierkomplement): ...11101100
int b = a >>> 2; // Binär: 00...00111011 (ein großer positiver Wert)
System.out.println(b);
```



Implementierung von Bitshifts ohne Operatoren

Bitshifts können mithilfe von logischen Operationen und Division durch 2 simuliert werden.

Left Shift (`<< n`) ohne Operator

Ein Left Shift um eine Stelle ist äquivalent zur Multiplikation mit 2. Ein Left Shift um `n` Stellen kann durch `n` Multiplikationen mit 2 erreicht werden.

```
int x = 5;
int n = 3;
int result = x;
for (int i = 0; i < n; i++) {
    result = result * 2;
```

```
}
```

```
System.out.println(result); // Ausgabe: 40 (5 << 3)
```

Right Shift (`>> n` oder `>>> n`) ohne Operator

Ein Right Shift um eine Stelle ist äquivalent zur Integer-Division durch 2. Ein Right Shift um `n` Stellen kann durch `n` Integer-Divisionen durch 2 erreicht werden.

Für den **Signed Right Shift (`>>`)** funktioniert die einfache Integer-Division für positive Zahlen. Bei negativen Zahlen ist zu beachten, dass die Integer-Division in Java immer in Richtung Null rundet. Um das Verhalten des Signed Right Shifts exakt nachzubilden, insbesondere bei negativen Zahlen, ist es komplexer und erfordert zusätzliche Logik, um das Vorzeichenbit korrekt zu behandeln.

Für den **Unsigned Right Shift (`>>>`)** ist die einfache Integer-Division nicht ausreichend, da sie das Vorzeichenbit beachtet. Um einen Unsigned Right Shift ohne den Operator zu implementieren, müsste man die Zahl als vorzeichenlose Größe behandeln (was in Java mit primitiven `int` nicht direkt möglich ist) und die Bits manuell verschieben und Nullen einfügen. Dies wäre deutlich aufwändiger und würde typischerweise bitweise logische Operationen erfordern, die den Shift-Operator im Grunde nachbilden.

Vereinfachte Simulation des Right Shifts (ähnlich `>>` für positive Zahlen):

```
int x = 20;
int n = 2;
int result = x;
for (int i = 0; i < n; i++) {
    result = result / 2; // Integer-Division
}
System.out.println(result); // Ausgabe: 5 (20 >> 2)
```



Wichtiger Hinweis: Die exakte Nachbildung des `>>>`-Operators ohne den Operator ist in Java mit reinen arithmetischen Operationen nicht trivial und würde den Einsatz bitweiser logischer Operationen erfordern, was den Sinn der Übung, den Operator zu vermeiden, untergräbt. Die gezeigte Division simuliert eher das Verhalten des `>>`-Operators für positive Zahlen.

Anwendungsfälle

Bitweise Shift Operatoren sind nützlich für:

- Effiziente Multiplikation und Division mit Zweierpotenzen.
- Manipulation einzelner Bits in Datenstrukturen (z.B. Flags, Bitssets).
- Low-Level-Programmierung und Hardware-Interaktion.
- Kompakte Speicherung von Zuständen.

Auf null setzen

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

Kurz gesagt, das explizite Setzen von Referenzen auf `null`, wenn sie nicht mehr benötigt werden, dient zwei Hauptzwecken:

1. **Vereinfachung der Logik:** Durch das Setzen auf `null` kann man sich in vielen Fällen Sonderbehandlungen für nicht gefundene oder ungültige Einträge sparen. Eine `null`-Referenz ist ein klar definierter Zustand.
2. **Verbesserung der Speicherverwaltung (indirekt):** Obwohl die Java Garbage Collection nicht sofort bei `null`-Zuweisung aktiv wird, signalisiert es dem Garbage Collector, dass das referenzierte Objekt potenziell freigegeben werden kann, da keine aktiven Referenzen mehr darauf existieren (außer möglicherweise im Debugger). Dies kann indirekt zur effizienteren Speichernutzung beitragen, insbesondere bei Objekten, die in langlebigen Datenstrukturen gehalten werden.

Es ist wichtig zu verstehen, dass das Setzen auf `null` nicht *immer* notwendig ist, aber in bestimmten Situationen (wie in dem genannten Beispiel mit Array-Einträgen) die Code-Wartbarkeit und potenziell die Speichernutzung verbessern kann.

Arrays vergrößern

```

public void addFirst(String e) {
    es[head = (head - 1) & mask] = e;
    if (tail == head) { doubleCapacity(); } — sofort verdoppeln wenn voll,
} — Zweierpotenz beibehalten

private void doubleCapacity() {
    mask = (mask << 1) | 1; — ein Bit mehr
    String[] newes = new String[mask + 1]; — neues Array
    int i = 0, j = 0; — gilt noch immer: tail == head
    while (i < head) { newes[j++] = es[i++]; }
    j = head += es.length; — ab hier: tail != head, Lücke nicht gefüllt
    while (i < es.length) { newes[j++] = es[i++]; }
    es = newes;
}

```

Zusammenfassend lässt sich sagen, dass der Code eine dynamisch vergrößerbare Datenstruktur implementiert, die das Hinzufügen von Elementen am Anfang optimiert, indem sie einen Ringpuffer und eine Verdopplungsstrategie für die Kapazität nutzt.

Das Array wird vergrößert, wenn beim Einfügen eines neuen Elements der `tail` den `head` erreicht.

Vergleich wenn null als Wert zählt

```

public class SimpleAssoc {
    private int top; — jeder Array-Index i mit i < top ist gültig
    private String[] ks = new String[8]; — zwei getrennte Arrays für Schlüssel und Werte
    private String[] vs = new String[8]; — kleine Zweierpotenz
    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 $i < \text{top}$ 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

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];
        }
        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;
}

```

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

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];
        ks[top] = null;
        vs[i] = vs[top];
        vs[top] = null;
    }
    return old;
}

```

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)]; 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  
}
```