# Algorithmen und Datenstrukturen

Stapel und Warteschlangen

Prof. Justus Piater, Ph.D.

16. März 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

## Inhaltsverzeichnis



Einführung

In diesem Kapitel werden wir unseren ersten abstrakten Datentyp im Detail betrachten, den sogenannten ***Stapel***, auf englisch *stack*. Diese Bezeichnung beschreibt anschaulich die beiden Operationen, die einen Stapel definieren: Man kann ein Element oben auf den Stapel legen, und man kann das oberste Element herunter holen. Zugriff auf Elemente ist jedoch nur oben auf dem Stapel möglich. Wir können nicht weiter unten im Stapel Elemente einfügen oder entfernen – wie bei einem Stapel Bauklötze.

Mit anderen Worten, das *erste* Element, das ich zu einem gegebenen Zeitpunkt herunter hole, ist immer das *letzte*, das ich vorher auf dem Stapel abgelegt habe. Dieses Funktionsprinzip wird auf englisch als *Last In, First Out*, bezeichnet, kurz LIFO.

Anschließend werden wir uns zwei weitere, verwandte abstrakte Datentypen anschauen. Die sogenannte ***Warteschlange***, englisch *queue*, folgt dem *First In, First Out*-Prinzip, abgekürzt FIFO. Wir können jederzeit Elemente hinten anstellen oder vorne entfernen, aber nicht dazwischen.

Kombinieren wir die Methoden des Stapels und der Warteschlange, erhalten wir den sogenannten ***Doppelstapel***, auf englisch *double-ended queue* oder kurz *deque*. Man könnte ihn genauso gut als Doppelschlange oder als *double-ended stack* bezeichnen, aber diese Begriffe sind nicht gebräuchlich. Bei einem Doppelstapel können wir jederzeit Elemente an einem beliebigen Ende einfügen oder entfernen, aber nicht dazwischen.

Für alle diese abstrakten Datentypen werden wir diskutieren, welche *Datenstrukturen* sich zu ihrer Implementierung eignen. Insbesondere werden wir verschiedene Formen von Arrays und verketteten Listen in Erwägung ziehen. Ihre jeweilige Eignung ergibt sich aus ihrem Platzbedarf sowie dem Zeitbedarf der Algorithmen, die die Methoden des gegebenen abstrakten Datentyps auf Basis der jeweiligen Datenstruktur implementieren. Wir werden also auch den asymptotischen Ressourcenbedarf dieser Algorithmen formal analysieren. Damit werden Sie in der Lage sein, für Ihre konkreten Programmierprobleme geeignete abstrakte Datentypen, Datenstrukturen und Algorithmen auszuwählen.

Nach dem gleichen Schema werden wir in vielen der folgenden Kapitel verfahren: Wir definieren abstrakte Datentypen durch ihre Methoden, und überlegen dann, mittels welcher Datenstrukturen und zugehörigen Algorithmen sich diese Methoden effizient implementieren lassen.

Damit sammeln Sie sich über den Verlauf dieses Kurses einen wertvollen Schatz aus abstrakten Datentypen, Datenstrukturen, Algorithmen und Analysemethoden an. Daraus können Sie sich dann bedienen, um effiziente Algorithmen für Programmierprobleme aus der Praxis zu entwickeln.

# 1 Stapel

Video 1 beginnt hier.

## ADT: Stapel (*stack*, LIFO) [Slide 1]

LIFO = *last in, first out*

```
push(e)    // Adds element e to the top of the stack.

pop()      // Removes and returns the top element
           // from the stack
           // (or null if the stack is empty).

// Accessor methods for convenience:

top()      // Returns the top element of the stack,
           // without removing it
           // (or null if the stack is empty).
size()     // Returns the number of elements
           // in the stack.
isEmpty()  // Returns a boolean indicating
           // whether the stack is empty.
```

[Curology auf Unsplash]

Anwendungen?

- Web-Browser-Verlauf
- Undo

Vgl. ADT: Warteschlange (*queue*, FIFO).

Definition

Der abstrakte Datentyp ***Stapel*** wird durch die Methoden `push()` und `pop()` definiert. `push(e)` legt das Element `e` oben auf den Stapel. Die Methode `pop()` entfernt das oberste Element vom Stapel und liefert es zurück. Das zuletzt gepushte Element wird also immer als Erstes gepoppt. Dieses Prinzip nennt sich auf englisch *Last In, First Out*, abgekürzt LIFO.

Neben `push()` und `pop()` definieren wir noch die Abfragemethode `top()`. Sie liefert das oberste Element des Stapels zurück, genau wie `pop()`, aber belässt es auf dem Stapel, im Gegensatz zu `pop()`.

Die drei Methoden `push()`, `pop()` und `top()` sind die charakteristischen Methoden des abstrakten Datentyps Stapel. Darüber hinaus definieren wir zwei weitere Methoden,

`size()` und `isEmpty()`. Diese beiden Methoden erleichtern uns den praktischen Umgang mit Datenstrukturen, und werden uns in vielen anderen abstrakten Datentypen wieder begegnen.

Erklärung

Die englischen Begriffe *push* und *pop* lassen sich mit der Analogie eines Tellerstapelwagens erklären, wie man sie in der Gastronomie verwendet. Sie kennen diese vielleicht: Der Tellerstapel steht auf einer gefederten Plattform, die er durch sein eigenes Gewicht so weit hinunter drückt, dass oben immer nur einige wenige Teller zugänglich sind, egal, wie hoch der Tellerstapel ist. Hier kann man einen weiteren Teller auflegen und damit den Stapel weiter hinunterdrücken, also *push the plate down onto the stack*. Umgekehrt stelle man sich vor, die gespannte Feder werfe den obersten Teller wieder aus, so dass *it pops back up from the stack*. Zum Glück tun Tellerstapelwagen dies meist nicht spontan!

Der Stapel gehört zu den am weitesten verbreiteten abstrakten Datentypen überhaupt. Wir haben ihn bereits im vorigen Kapitel kennen gelernt, ohne den Begriff zu benutzen: Für jeden Funktionsaufruf wird für die Argumente, lokale Variablen und andere Daten des Funktionsaufrufs ein Speicherbereich reserviert. Dieser Speicherbereich nennt sich der *stack frame* dieses Funktionsaufrufs, und er wird auf den Laufzeitstapel des aktuellen *Threads* geschoben, englisch *runtime stack*. Verschachtelte Funktionsaufrufe pushen jeweils ihren *stack frame* und poppen ihn beim Rücksprung.

Ein weiteres Anwendungsbeispiel ist die *Undo*-Funktion in Ihrer Textverarbeitung. Während Sie schreiben, fasst Ihre Textverarbeitung kleinere Gruppen zusammenhängender Änderungen zusammen und schiebt sie auf den *Undo Stack*. Wenn Sie nun den *Undo*-Knopf klicken, dann wird die jüngste Gruppe Änderungen vom *Undo Stack* gepoppt, und diese Änderungen werden im Text rückgängig gemacht.

Durch wiederholtes Klicken von *Undo* können Sie Ihren Text weiter in die Vergangenheit zurücktransformieren. Sie können jedoch nicht ohne Weiteres bestimmte Änderungen tief in der Vergangenheit rückgängig machen, ohne dabei die jüngere Vergangenheit anzutasten, da spätere Änderungen in der Regel auf früheren aufbauen. Sie müssen also den Stack Element für Element poppen.

Bis jetzt haben wir lediglich darüber gesprochen, welche Funktionen der Stapel unterstützt, nämlich im Wesentlichen `push()` und `pop()`. Diese definieren den abstrakten Datentyp. Wir haben noch nichts darüber gesagt, in welchen Datenstrukturen wir die Daten ablegen wollen, und welche Algorithmen `push()`, `pop()` und die anderen Methoden implementieren sollen.

## Quiz [Slide 2]

Die Methoden `top()`, `size()` und `isEmpty()` sind…

A: vollkommen überflüssig; man könnte sie ebenso gut mittels `push()` und `pop()` implementieren.

B: hilfreich, obwohl man sie auch mittels `push()` und `pop()` implementieren könnte.

C: notwendig.

D: weiß nicht

### Stack: Interface in Java [Slide 3]

***Anmerkung***

- Java bietet eine Klasse `java.util.Stack`, aber sie ist obsolet, da sie mit dem moderneren *Java Collections Framework* inkonsistent ist. Außerdem unterscheidet sich ihre Fehlerbehandlung von unserem ADT.

  Ähnliches gilt für viele andere Interfaces dieses Kurses.

- Die Notation `<E>` bezeichnet hier einen *generischen Datentyp*. Hier garantiert er, dass dieses Interface sich auf jeden beliebigen Datentyp beziehen kann, und dass `top()` und `pop()` jeweils ein Element *desselben Typs* zurückgeben, wie `push()` empfängt.

```
/**
 * A collection of objects that are inserted and removed according to the last-in
 * first-out principle. Although similar in purpose, this interface differs from
 * java.util.Stack.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public interface Stack<E> {

  /* Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  int size();

  /* Tests whether the stack is empty.
   * @return true if the stack is empty, false otherwise
   */
  boolean isEmpty();

  /* Inserts an element at the top of the stack.
   * @param e the element to be inserted
   */
  void push(E e);

  /* Returns, but does not remove, the element at the top of the stack.
   * @return top element in the stack (or null if empty)
   */
  E top();

  /* Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  E pop();
}
```

## Implementation: Array [Slide 4]

Video 2 beginnt hier.

Animation

Eine Datenstruktur, die sich für Stapel eignet, ist das Array. Für unsere Zwecke besteht diese Datenstruktur aus dem Array selbst sowie einer Variablen, die die Anzahl der Elemente im Stack enthält, hier `size` genannt. Dies ist gleichzeitig der Index des ersten freien Felds im Array.

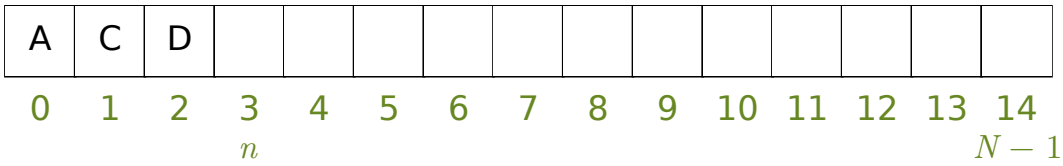
Pushen wir nun das Element A. Es wird in dem Feld mit Index `size` abgelegt, und anschließend wird `size` inkrementiert. Das gleiche passiert, wenn wir nun ein zweites Element pushen, hier B.

Ein Aufruf von `pop()` verläuft genau umgekehrt. Zunächst wird `size` dekrementiert, und anschließend wird das Element am Index `size` entfernt und zurückgeliefert.

Pushen und poppen wir noch ein wenig. Wir sehen, dass die Elemente immer am Anfang des Arrays konzentriert bleiben.

Was machen wir, wenn das Array voll ist? Hier beschränken wir einfach seine Größe und geben einen Fehler aus, wenn `push()` auf einem vollen Array aufgerufen wird. Man kann auch das Array vergrößern. Das werden wir uns im nächsten Kapitel anschauen.

## Implementation: Array [Slide 5]




$n$ = Zahl der Elemente = erster leerer Index

```
Algorithm size():
  return n

Algorithm isEmpty():
  return n = 0

Algorithm top():
  if isEmpty() then
    return null
  return s[n − 1]
```

```
Algorithm push(o):
  if size()= N then
    error
  s[n] ← o
  n ← n + 1

Algorithm pop():
  if isEmpty() then
    return null
  n ← n − 1
  return s[n]
```

Erklärung

Sehen wir hier die Algorithmen aller fünf Methoden des abstrakten Datentyps Stapel für eine Array-Datenstruktur. Die Variable `size` aus der Animation heißt hier etwas abstrakter klein-$n$, und die maximale Kapazität des Arrays ist groß-$N$.

Wir sehen, dass `push()` und `pop()` genau so funktionieren wie in der Animation beschrieben, und die anderen drei Methoden sind trivial.

Was ist der Ressourcenbedarf unserer Array-basierten Implementierung eines Stapels?

Wenn wir uns die Laufzeit der einzelnen Anweisungen der fünf Algorithmen anschauen, dann sehen wir, dass keine von klein-$n$ oder von groß-$N$ abhängt. Damit ist die Laufzeit aller fünf Methoden konstant, also $O(1)$.

Wie sieht es mit dem asymptotischen Platzbedarf aus?

Man könnte meinen, der Platzbedarf sei $\Theta(n)$. Dem ist jedoch nicht so, denn das Array

> belegt einen Platz proportional zu groß-$N$, unabhängig von der Zahl klein-$n$ der Elemente im Stapel.
>
> Ist die Kapazität groß-$N$ durch den Anwender wählbar, dann kann man den Platzbedarf sinnvollerweise als $\Theta(N)$ beschreiben. Wird das Array je nach Bedarf automatisch vergrößert, dann ist der Platzbedarf $\Theta(n)$. Ist groß-$N$ jedoch de facto konstant, dann hängt der Platzbedarf von überhaupt nichts ab, und ist damit $O(1)$.

## Komplexität [Slide 6]

- Zeit:

  Alle Methoden $O(1)$

- Platz:

  $O(1)$ – begrenzte Kapazität!

### Stack: Array-basierte Implementation in Java [Slide 7]

```
/**
 * Implementation of the stack ADT using a fixed-length array. All
 * operations are performed in constant time. An exception is thrown
 * if a push operation is attempted when the size of the stack is
 * equal to the length of the array.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000; // default array capacity
  private E[] data;                      // generic array used for storage
  private int t = -1;                    // index of the top element in stack
  public ArrayStack() { this(CAPACITY); } // constructs stack with default capacity

  /**
   * Constructs an empty stack with the given array capacity.
   * @param capacity length of the underlying array
   */
  public ArrayStack(int capacity) {  // constructs stack with given capacity
    data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
  }

  /**
   * Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  public int size() { return t + 1; }

  /**
   * Tests whether the stack is empty.
   * @return true if the stack is empty, false otherwise
   */
  public boolean isEmpty() { return t == -1; }

  /**
   * Inserts an element at the top of the stack.
   * @param e the element to be inserted
   * @throws IllegalStateException if the array storing the elements is full
   */
  public void push(E e) throws IllegalStateException {
    if (size() == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                       // increment t before storing new item
  }

  /**
   * Returns, but does not remove, the element at the top of the stack.
   * @return top element in the stack (or null if empty)
   */
```

```
  public E top() {
    if (isEmpty()) return null;
    return data[t];
  }

  /**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  public E pop() {
    if (isEmpty()) return null;
    E answer = data[t];
    data[t] = null;                    // dereference to help garbage collection
    t--;
    return answer;
  }
}
```

## Implementation: einfach verkettete Liste [Slide 8]

Animation

Eine andere geeignete Datenstruktur für Stapel ist die verkettete Liste. Wird das erste Element gepusht, hier A, wird es in einem neuen Knoten abgelegt, dessen Zeiger auf den Nachfolger leer ist, und der Kopf-Zeiger wird auf diesen Knoten gesetzt.

Beim Pushen weiterer Elemente wird das neue Element wiederum in einem neuen Knoten abgelegt, dessen Folgezeiger auf den Wert des aktuellen Kopf-Zeigers gesetzt wird. Anschließend wird der Kopf-Zeiger auf den neuen Knoten gesetzt.

Um ein Element zu poppen, wird das Element des Knotens zurückgeliefert, auf das der Kopf-Zeiger zeigt. Der Kopf-Zeiger wird anschließend auf den Wert des Folgezeigers dieses Knotens gesetzt. Der durch diesen ausgehängten Knoten belegte Speicherbereich kann nun freigegeben werden.

Auf diese Weise wächst und schrumpft die Liste mit jedem `push()` und `pop()`, und der Kopf der Liste zeigt stets auf das oberste Element des Stapels.

Erklärung

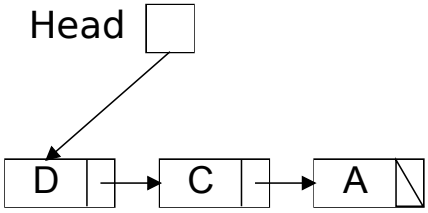
Was ist die Laufzeit-Komplexität der Algorithmen für die fünf Methoden des abstrakten Datentyps Stapel?

Wir haben diese Algorithmen hier nicht ausformuliert, aber man kann sich ihre Schritte leicht überlegen. Insbesondere bestehen alle Schritte von `push()` und `pop()` darin, Zeiger einen Schritt weit zu verfolgen oder Zeiger umzuhängen. An keiner Stelle spielt die Länge der Liste eine Rolle. Daher lassen sich `push()`, `pop()` und trivialerweise auch `top()` und `isEmpty()` in konstanter Laufzeit implementieren.

Wie implementieren wir die verbleibende Methode, `size()`, die die Anzahl der Elemente im Stapel zurück gibt? Eine naheliegende Methode besteht darin, die Liste zu traversieren und dabei die Elemente zu zählen. Dies führt jedoch zu einer linearen Laufzeit in der Anzahl $n$ der Elemente auf dem Stapel. Um die Methode `size()` in konstanter Laufzeit implementieren zu können, wird diese Datenstruktur daher üblicherweise um eine Variable $n$ ergänzt, deren Wert von der Methode `size()` in konstanter Zeit zurückgegeben werden kann. Diese Variable muss von `push()` und `pop()` inkrementiert bzw. dekrementiert werden. Das hat jedoch keinerlei Auswirkungen auf die asymptotische Laufzeit.

Dies ist ein einfaches Beispiel dafür, dass geeignete Datenstrukturen entscheidende Verbesserungen im Laufzeitverhalten ermöglichen können. Hier reduziert die zusätzliche Variable $n$ die Laufzeit der Methode `size()` von $\Theta(n)$ auf $O(1)$, ohne dass dafür irgendein asymptotischer Preis gezahlt werden muss.

### Implementation: einfach verkettete Liste [Slide 9]




Komplexität der Methoden?

### Anwendung: Klammern paaren [Slide 10]

```
Algorithm parenMatch(X, n):
  Require: a string X of n parentheses
           of different types.
  Ensure: Return true iff all the parentheses match.
  s ←empty stack
  for i ← 0 to n − 1 do
    if X[i] is a left parenthesis then
      s.push(X[i])
    else
      if s.isEmpty() then
        return false
      if s.pop() does not match X[i] then
        return false
  if s.isEmpty() then
    return true
  else
    return false
```

Korrekt:

- ()(()){([()])}
- (((()(()){([()])}))

Inkorrekt:

- )(()){([()])}
- ({[])}
- (

# 2 Warteschlangen

Video 3 beginnt hier.

## ADT: Warteschlange (*queue*, FIFO) [Slide 11]

FIFO = *First in, first out*

```
enqueue(e)  // Adds element e to the back of the queue.

dequeue()   // Removes and returns the first element
            // from the queue
            // (or null if the queue is empty).

// Accessor methods for convenience:

first()     // Returns the first element of the queue,
            // without removing it
            // (or null if the stack is empty).
size()      // Returns the number of elements
            // in the queue.
isEmpty()   // Returns a boolean indicating
            // whether the queue is empty.
```

[falco auf Pixabay]

Anwendungen?

Vgl. ADT: Stapel (*stack*, LIFO).

Definition

Der abstrakte Datentyp ***Warteschlange*** ist das *first in, first out*-Gegenstück zum *last in, first out*-Stapel. Die Warteschlange heißt auf englisch *Queue*. Entsprechend heißen die beiden definierenden Methoden `enqueue()` und `dequeue()`. `enqueue()` ist das Gegenstück der Methode `push()` des Stapels, fügt jedoch am Ende ein und nicht am Anfang. `dequeue()` entnimmt vorne das erste Element und ist äquivalent zur Methode `pop()` des Stapels. Ebenso ist `first()` äquivalent zur Methode `top()` des Stapels und heißt hier lediglich anders, um bei der Warteschlangen-Analogie zu bleiben.

Erklärung

Eine Warteschlange funktioniert wie ein Förderband: Pakete werden an einem Ende aufgelegt und am anderen Ende in derselben Reihenfolge wieder abgenommen. Dazwischen kann man keine Pakete einfügen oder herausnehmen, und man kann auch nicht die Richtung ändern.

Wie Stacks gehören auch Queues zu den wichtigsten abstrakten Datentypen mit Anwendungen in vielen Bereichen. Ein Beispiel ist ein Web-Server: Vom Internet treffen Anfragen ein und werden hinten in eine Queue eingereiht. Am vorderen Ende der Queue holen sich Prozessoren des Servers die jeweils nächste Anfrage.

## Queue: Interface in Java [Slide 12]

```
/**
 * Interface for a queue: a collection of elements that are inserted
 * and removed according to the first-in first-out principle. Although
 * similar in purpose, this interface differs from java.util.Queue.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public interface Queue<E> {
  /* Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  int size();

  /* Tests whether the queue is empty.
   * @return true if the queue is empty, false otherwise
   */
  boolean isEmpty();

  /* Inserts an element at the rear of the queue.
   * @param e the element to be inserted
   */
  void enqueue(E e);

  /* Returns, but does not remove, the first element of the queue.
   * @return the first element of the queue (or null if empty)
   */
  E first();

  /* Removes and returns the first element of the queue.
   * @return element removed (or null if empty)
   */
  E dequeue();
}
```

## Implementation: zyklisches Array [Slide 13]

Animation

Ähnlich wie Stapel lassen sich Warteschlangen als Arrays implementieren. Das Einfügen mittels `enqueue()` funktioniert genauso wie `push()` beim Stapel. Hier rufen wir `enqueue()` auf den Elementen A und B auf. Anschließend rufen wir `dequeue()` auf. Im Gegensatz zum Stapel liefert uns dies nicht das letzte, sondern das erste Element des Arrays.

Nun könnte man den verbleibenden Inhalt des Arrays nach links verschieben, um die entstandene Lücke aufzufüllen. Dies würde jedoch eine Laufzeit von $\Theta(\text{size})$ erfordern. Stattdessen setzen wir eine Hilfsvariable `front` ein, die den Index des ersten Elements der Warteschlange enthält. Dieser Index wird durch `dequeue()` inkrementiert. Nun muss nur noch die Variable `size` dekrementiert werden, die die Anzahl der Elemente in der Warteschlange enthält.

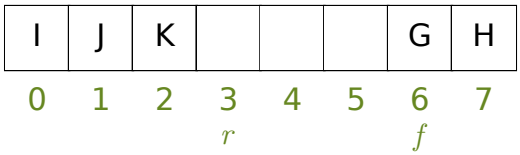
Wir sehen, dass jedes `dequeue()` den Index des ersten Elements nach rechts verschiebt

und somit im Array immer weniger Platz für folgende Elemente verbleibt. Nachdem wir das Element H eingefügt haben, ist das Array am Ende, obwohl `size` nur gleich 5 ist, bei einer Kapazität von 8. Es sind also drei Plätze vorn im Array frei.

Wir machen diese nutzbar, indem wir das Array als *zyklisch* auffassen. Auf den Index 7 folgt der Index 0; das Array hat keinen Anfang und kein Ende. Hierzu wird der für `enqueue()` berechnete Index modulo der Kapazität des Arrays genommen, hier als $N$ bezeichnet.

Wenn wir nun ein weiteres Element I einfügen, ist `front + size` $= 8$, modulo 8 gleich 0, und I wird am Anfang des Arrays eingefügt. Auf diese Weise können wir die volle Kapazität des Arrays nutzen.

## Implementation: zyklisches Array [Slide 14]




$N = 8$
$n = 5$
$r = (f + n) \ \% \ N$

$f$ (*front*) ist der Index des Anfangs der Warteschlange, $r$ (*rear*) zeigt hinter das Ende.

```
Algorithm size():
  return n

Algorithm isEmpty():
  return size() = 0

Algorithm front():
  if isEmpty() then
    return null
  return q[f]
```

```
Algorithm dequeue():
  if isEmpty() then
    return null
  t ← q[f]
  f ← (f + 1) mod N
  n ← n − 1
  return t

Algorithm enqueue(o):
  if size() = N then
    error
  q[(f + n) mod N] ← o
  n ← n + 1
```

Erklärung

Werfen wir noch einen kurzen Blick auf die Algorithmen. Wie beim Stapel heißt die Kapazität hier groß-$N$ und die Anzahl der Elemente klein-$n$. Die Variable $f$ steht für `front`, und die Hilfsvariable $r$ für *rear*, also den ersten freien Index am hinteren Ende der Warteschlange.

Hinsichtlich des Ressourcenbedarfs ist die Argumentation dieselbe wie beim Array-basierten Stapel. Vergewissern Sie sich selbst, dass die Laufzeiten aller Algorithmen konstant sind.

Das zyklische Array ist ein weiteres Beispiel dafür, wie man eine Datenstruktur so anpassen kann, dass sich das Laufzeitverhalten gewisser Funktionen deutlich verbessert. Im Vergleich zu einer Version von `dequeue()`, die den Arrayinhalt an den linken Anschlag verschiebt, verbessert das zyklische Array die Laufzeit von $\Theta(n)$ auf $O(1)$. Der Preis sind die zusätzliche Variable $f$ und einige Modulus-Operationen, aber diese haben keinerlei Einfluss auf den asymptotischen Ressourcenbedarf.

### Komplexität [Slide 15]

- Zeit:

  Alle Methoden $O(1)$

- Platz:

  $O(1)$ – begrenzte Kapazität!

### Implementation: einfach verkettete Liste [Slide 16]

Animation

Bei einer verketteten Liste als Datenstruktur für den abstrakten Datentyp der Warteschlange werden neue Elemente am Ende der Liste eingefügt statt am Anfang wie beim Stapel. Damit das Ende in konstanter Zeit erreichbar ist, wird extra ein `tail`-Zeiger verwendet.

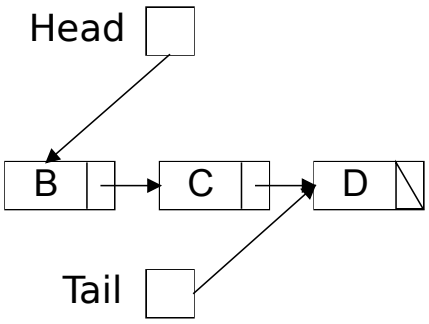
Wir sehen die allgemeine Prozedur beim Einfügen des zweiten Elements, B. Der letzte Knoten der Liste wird in konstanter Zeit über den `tail`-Zeiger erreicht. Dessen Folgezeiger wird auf den neuen Knoten mit dem neuen Element gesetzt, ebenso wie der `tail`-Zeiger selbst.

`dequeue()` funktioniert genauso wie `pop()` beim Stapel.

Zur Charakterisierung des Laufzeitverhaltens sehen wir wiederum, dass keine der Operationen von `enqueue()` und `dequeue()` von der Länge der Liste abhängen. Für die effiziente Unterstützung der `size()`-Methode können wir wie beim Stapel eine Variable $n$ einführen, die von `enqueue()` und `dequeue()` inkrementiert bzw. dekrementiert wird. Damit lassen sich alle fünf Methoden der Warteschlange in konstanter Laufzeit implementieren.

Für diese effiziente Lösung ist jedoch die Richtung der Liste entscheidend: `enqueue()` hängt am Ende an, und `dequeue()` entfernt am Kopf der Liste. Überlegen Sie selbst, wie Sie diese beiden Methoden mit einer umgekehrten Liste implementieren würden, und was ihre asymptotischen Laufzeiten wären.

### Implementation: einfach verkettete Liste [Slide 17]




Komplexität der Methoden?

# 3 Doppelstapel

Video 4 beginnt hier.

## ADT: Doppelstapel (*deque; double-ended queue*) [Slide 18]

Generalisierung von sowohl Stack als auch Queue.

```
addFirst(e)   // Inserts a new element e at the front of the deque.
addLast(e)    // Inserts a new element e at the rear of the deque.
removeFirst() // Removes and returns the front element of the deque
              // (or null if the deque is empty).
removeLast()  // Removes and returns the rear element of the deque
              // (or null if the deque is empty).

// Accessor methods for convenience:

first()   // Returns the front element of the deque, without removing it
          // (or null if the deque is empty).
last()    // Returns the rear element of the deque, without removing it
          // (or null if the deque is empty).
size()    // Returns the number of elements in the deque.
isEmpty() // Returns a boolean indicating whether the deque is empty.
```

Anwendungen?

Lastverteilung eines Web-Dienstes: Anfragen werden `insertLast()`, Server `removeFirst()` sie, und `insertFirst()` ihre Teilanfragen.

Definition

Der ***Doppelstapel*** ist eine Kombination von Stapel und Warteschlange, die das Einfügen und Entfernen an beiden Enden erlaubt. Auf englisch nennt er sich *double-ended queue*, üblicherweise abgekürzt mit dem Kunstwort *deque*.

Achtung: Deque schreibt sich ähnlich wie *dequeue*, aber spricht sich völlig anders aus und bedeutet auch etwas völlig anderes.

Der Doppelstapel bietet uns jeweils zwei Methoden zum Einfügen und Entfernen, `addFirst()`, `addLast()`, `removeFirst()`, und `removeLast()`, sowie zwei Abfragemethoden `first()` und `last()`.

Die Antisymmetrie zwischen den Bezeichnungen *first* und *last* dient lediglich der Unterscheidung der beiden Enden des Doppelstapels. Ein Doppelstapel verhält sich jedoch komplett symmetrisch.

Erklärung

Doppelstapel treten in der Praxis seltener auf als Stapel oder Warteschlangen. Ein mögliches Anwendungsbeispiel ist ein lastausgleichender Webserver. Manche Anfragen lassen sich in mehrere unabhängige Teilanfragen zerlegen, beispielsweise bei einer Sammelanfrage mehrerer Tracking-Nummern auf der Website eines Zustelldienstes.

Der Prozessor, der eine solche Sammelanfrage bearbeitet, schiebt diese Teilanfragen mittels `addFirst()` in seinen Doppelstapel. Dort holt er sie eine nach der anderen mittels `removeFirst()` wieder heraus und arbeitet sie ab. Er behandelt den Doppelstapel also wie einen gewöhnlichen Stapel.

In der Zwischenzeit jedoch können andere Prozessoren, die gerade nichts zu tun haben, Teilanfragen mittels `removeLast()` vom anderen Ende des Doppelstapels anderer Prozessoren stehlen, ohne den Eigentümer bei seiner Arbeit zu stören.

Die Implementierung des Doppelstapels auf Basis eines Arrays sollte für Sie nun eine einfache Übung sein, ebenso wie die Analyse des Ressourcenbedarfs.

> Bei der Implementierung auf Basis einer verketteten Liste haben wir bereits gesehen, dass für effiziente Algorithmen die Richtung der Verkettung entscheidend ist. Da der Doppelstapel jedoch symmetrisch ist, muss auch die verkettete Liste symmetrisch sein. Wir benötigen daher eine doppelt verkettete Liste. Die Details können Sie sich selbst überlegen.

### Implementation als zyklisches Array [Slide 19]

Komplexität der Methoden?

***Anmerkung***

`addFirst()` und `removeLast()` erfordern zyklisches Dekrementieren. Je nach Semantik der verwendeten Programmiersprache verwende man $(f-1+N)\,\%\,N$ anstatt $(f-1)\,\%\,N$.

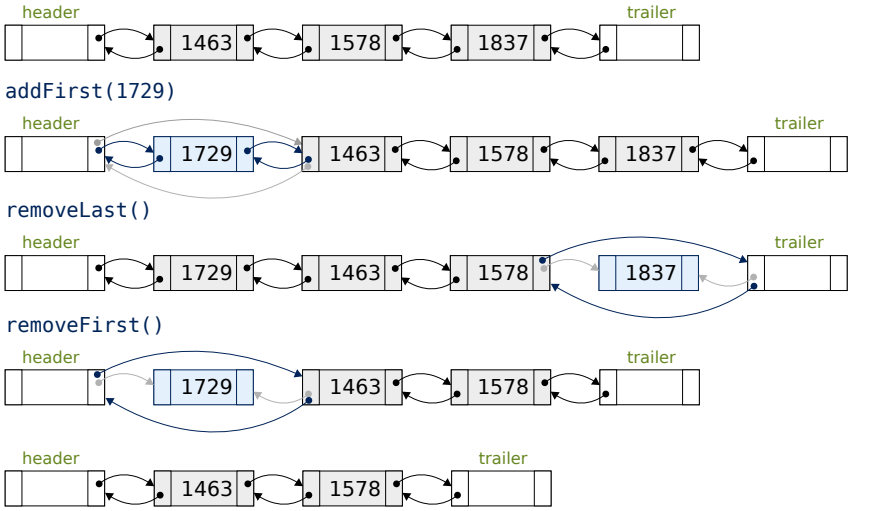
### Implementation als einfach verkettete Liste [Slide 20]

Komplexität der Methoden?

### Implementation als doppelt verkettete Liste [Slide 21]

Komplexität der Methoden?

### Doppelt verkettete Liste mit Sentinel Nodes [Slide 22]




***Anmerkung***

Die *Sentinel Nodes* `header` und `trailer` sorgen dafür, dass auch der erste Knoten einen Vorgänger und der letzte Knoten einen Nachfolger haben. Dadurch sind Einfüge- und Entfern-Operationen überall dieselben, egal ob am Anfang, in der Mitte, oder am Ende der Liste. Es müssen keine Ausnahmen berücksichtigt werden, was den Code vereinfacht.

# 4 Zusammenfassung

## Zusammenfassung [Slide 23]

Fundamentale Konzepte:

**ADT:**

- Stack + Queue = Deque

**DS:**

- Einfache and zyklische Arrays
- Einfach und doppelt verkettete Listen