



Algorithmen und Datenstrukturen

Kapitel 05: Abstrakte Datentypen und verkettete Listen

Prof. Dr. Adrian Ulges

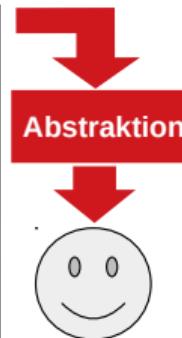
B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Abstraktion in der Informatik ist wichtig!

xxx

“Computer Science: The Mechanization of Abstraction”

(Aho, Ullmann: Foundations
of Computer Science)



Typisches Vorgehen (*Bsp. Routenplaner*)

1. Bilde ein (abstraktes) **formales Modell**
2. Transformiere es in ein „Maschinenmodell“.

Beispiel: Algorithmus → Programm

1. Algorithmus = **abstrakte Vorgehensbeschreibung** ('=' oder ':=' ? egal!)
2. **Programm** = konkrete Implementierung.

Beispiel: Anforderungsanalyse → Software-System

1. **Pflichtenheft** = Abstrakte Gliederung des Systems
2. Abbildung in **Software**.



Abstrakte Datentypen

Während Algorithmen **Verfahren** abstrahierten, abstrahieren wir nun **Daten**.

Abstrakte Datentypen

- ▶ Abstrakte Datentypen (ADTs) = **Spezifikation** von Daten
- ▶ **Formale Beschreibung** von Aufbau und Operationen des Datentyps
- ▶ Implementierung erfolgt dann später in **konkreten** Datentypen.

Informatiker...

- ▶ ... sollten in ADTs **“denken”** (*boolesche Logik, Arrays, Stacks, Hashmaps, ...*)
- ▶ Egal welche Sprache modern wird, ATDs wird es weiter geben! → unser Wissen bleibt also anwendbar ☺.

ADTs: Formalisierung

1. verwendete Typen
2. Operatoren
3. Axiome (=Grundannahmen, das “eigentliche” Verhalten).



Outline

1. Der Abstrakte Datentyp "Stack"

2. Stack: Implementierung

3. Dynamische Datenstrukturen

4. Verkettete Listen

5. Doppelt Verkettete Listen

Der Datentyp “Stack”

Der **Stack** (dt. *Stapel/Keller*) ist ein einfacher **Container-Datentyp** in Analogie zu einem **Stapel von Objekten**. Wir können ...

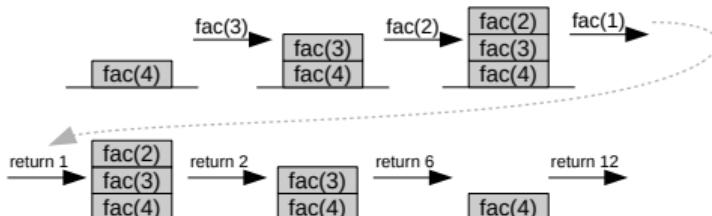
- ▶ ein Objekt **oben** auf den Stapel legen.
- ▶ das **oberste** Objekt vom Stapel nehmen.

Anwendungen

- ▶ Unterprogrammaufrufe (“*Stack Overflow*”)
- ▶ Syntax-Parsing in Compilern
- ▶ ...

```
def fac(n):  
    if n == 1:  
        return 1  
    else:  
        return fac(n-1)*n  
  
print fac(4)
```

Der Call-Stack, wenn wir
fac(4) aufrufen...



Beispiel (Stack<int>)

Stacks operieren nach dem sogenannten **LIFO-Prinzip** (engl. *Last-In-First-Out*):
Was **zuletzt** auf den Stack gelegt wurde, wird als **erstes** wieder entfernt.

Anweisungsfolge	Zustandsfolge
<code>S = empty</code>	 _____ → true
<code>is_empty(S)</code>	
<code>S = push(S,11)</code>	 _____ 11 → true
<code>S = push(S,7)</code>	 _____ 11 7 5 → true
<code>S = push(S,5)</code>	 _____ 11 7 5 → true
<code>top(S)</code>	 _____ 7 11 → 5
<code>S = pop(S)</code>	 _____ 11 → 7
<code>top(S)</code>	 _____ 11 → 7
<code>is_empty(S)</code>	_____ → false

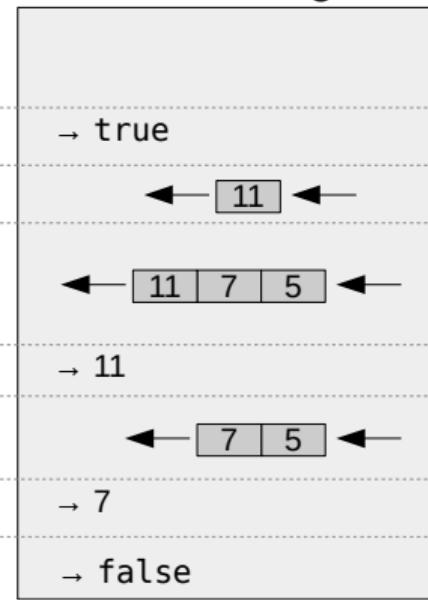
Queues (Warteschlangen)

Das “Gegenteil” eines Stacks ist eine Queue / Warteschlange. Hier gilt das **FIFO-Prinzip** (engl. *First-In-First-Out*): Was **zuerst** in die Queue geschoben wurde, wird als **erstes** wieder entfernt.

Anweisungsfolge

```
q = empty  
is_empty(q)  
q = push(q,11)  
q = push(q,7)  
q = push(q,5)  
top(q)  
q = pop(q)  
top(q)  
is_empty(q)
```

Zustandsfolge



Der ADT Stack<T>: Spezifikation

1. Verwendete Typen

- ▶ T (Typ der zu speichernden Objekte, vgl. Java Generics)
- ▶ Boolean

2. Operatoren

$$f: \mathbb{R} \rightarrow \mathbb{R}$$

empty: $\rightarrow \text{Stack}$ // Konstruktor, leerer Stack

push: $\text{Stack} \times T \rightarrow \text{Stack}$ // Element oben auf den Stack legen

top: $\text{Stack} \rightarrow T$ // oberstes Element zurückgeben

pop: $\text{Stack} \rightarrow \text{Stack}$ // oberstes Element entfernen

is_empty: $\text{Stack} \rightarrow \text{Boolean}$ // Test ob Stack leer.

Anmerkungen

- ▶ In Java werden Funktionen zu *Instanzvariablen* und *Methoden*.
- ▶ **Beispiel:** Aus s = push(s, e) wird in Java s.push(e).

ADT "Stack": Spezifikation II

Wie spezifizieren wir das **Verhalten** eines Stacks **formal**?

3. Axiome

Die folgenden Axiome gelten für alle Stacks:

1. `is_empty(empty) = true`

empty ist leerer Stack.

2. `is_empty(push(s,e)) = false`

Wenn als letztes ein Element auf Stack gepusht wurde,
ist dieser nicht leer.

3. `top(push(s,e)) = e`

Wenn ein Element e auf Stack gepusht wird,
dann ist es das oberste.

4. `pop(push(s,e)) = s`

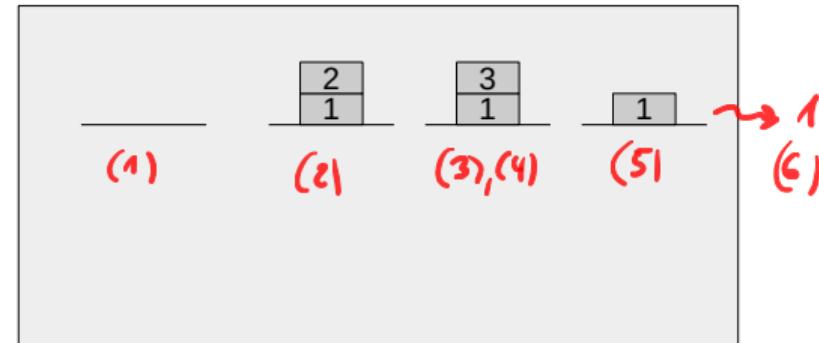
Wenn Elt. auf Stack gepusht wird und dann wieder
entnommen wird, erhalten wir wieder den vorherigen Stack.

```
type Stack(T)
import Bool
operators
    empty: -> Stack
    push: Stack × T -> Stack
    pop: Stack -> Stack
    top: Stack -> T
    is_empty: Stack -> Bool
axioms
    is_empty(empty) = true
    is_empty(push(s,x)) = false
    pop(push(s,x)) = s
    top(push(s,x)) = x
```

ADT "Stack": Beispiel-Verhalten

Die Axiome stellen die LIFO-Eigenschaften des Stacks sicher:

- 1 $S = \text{empty};$
- 2 $S = \text{push}(S, 1); S = \text{push}(S, 2);$
- 3 $S = \text{pop}(S);$
- 4 $S = \text{push}(S, 3);$
- 5 $S = \text{pop}(S);$ ~~$\text{push}(S, 4);$~~
- 6 $\text{top}(S) \rightarrow ?$



$\text{top}(\text{pop}(\text{push}(\text{pop}(\text{push}(\text{push}(\text{empty}, 1), 2)), 3)))$

Axiom 4

$\text{top}(\text{pop}(\text{push}($

Axiom 4

$\text{top}($

$\text{push}(\text{empty}, 1), 3)))$

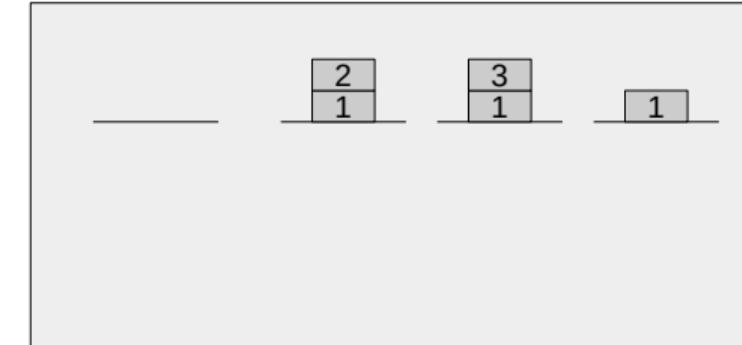
$\text{push}(\text{empty}, 1) = 1$ ✓

Axiom 3



ADT "Stack": Beispiel-Verhalten

```
S = empty;  
S = push(S,1); S = push(S,2);  
S = pop(S);  
S = push(S,3);  
S = pop(S); empty,  
top(S) → ?
```





Outline

1. Der Abstrakte Datentyp "Stack"

2. Stack: Implementierung

3. Dynamische Datenstrukturen

4. Verkettete Listen

5. Doppelt Verkettete Listen

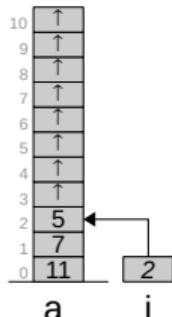
Stacks in Java

Erste Umsetzung von **Stacks in Java**

- Wir definieren eine **Schnittstelle** (*engl. Interface*) auf Basis des ADTs.
- Diese ergänzen wir durch eine Implementierung des Verhaltens (*Klasse*).
- Die Klasse verwendet ein **Array** zur Speicherung der Objekte.
- Eine **Indexvariable i** speichert die Position des obersten Elements.

```
public interface Stack<T> {  
    public void push(T e);  
    public void pop();  
    public T top();  
    public boolean isEmpty();  
}
```

```
class ArrayStack<T>  
    implements Stack<T> {  
    ...  
}
```



Operationen	... im Array
empty:	i = -1
is_empty:	i == -1
push:	a[++i] = e
pop:	i--
top:	a[i]

Beispiel
// Stack-OPs

```
s.empty(); → i = -1;  
s.push(11); → a[++i] = 11;  
s.push(7); → a[++i] = 7;  
s.push(5); → a[++i] = 5;  
s.pop(); → i--;
```

Beispiel
// Array-OPs

Java-Stack: Implementierung

- ▶ Die Klasse **ArrayStack** **implementiert** die Schnittstelle **Stack** (*Schlüsselwort 'implements'*). Alle Methoden (*Prototypen*) des Interfaces müssen implementiert werden!
- ▶ Array vom Typ **Object []**: Kann Objekte eines **beliebigen Typs T** speichern (*Object = allgemeinste Oberklasse*).

```
public class ArrayStack<T>
    implements Stack<T> {

    private Object[] a;
    private int i;

    public ArrayStack() { // empty
        a = new Object[1000];
        i = -1;
    }

    public void push(T e) {
        a[++i] = e;
    }

    public void pop() {
        a[i] = null;
        i--;
    }

    public T top() {
        return (T)a[i];
    }

    public boolean isEmpty() {
        return i== -1;
    }

}
```

Java-Stack: Implementierung

Wir haben ein **Interface Stack<T>** definiert.

Was ist der **Vorteil**?

Austauschbarkeit

- Wir können später ArrayStack durch andere Klassen ersetzen, ohne aufrufenden Code **anzupassen!**
- **Grundregel:** Immer gegen die allgemeinste Schnittstelle implementieren.

```
// Schlecht wäre:  
// ArrayStack<String> s = ...  
  
Stack<String> s =  
    new ArrayStack<String>();  
  
boolean b = s.isEmpty();  
s.push("Winter");  
s.push("is");  
...
```

```
public class ArrayStack<T>  
    implements Stack<T> {  
  
    private Object[] a;  
    private int i;  
  
    public ArrayStack() { // empty  
        a = new Object[1000];  
        i = -1;  
    }  
  
    public void push(T e) {  
        a[++i] = e;  
    }  
  
    public void pop() {  
        a[i] = null;  
        i--;  
    }  
  
    public T top() {  
        return (T)a[i];  
    }  
  
    public boolean isEmpty() {  
        return i == -1;  
    }  
}
```

Java-Stack: Implementierung

```
boolean isEmpty() {  
    return i == -1;  
}  
  
// Schlecht  
if (s.i == -1) ...  
  
// Gut  
if (s.isEmpty()) ...
```

Geheimnisprinzip

- ▶ Zur Austauschbarkeit darf der verwendende Code **nicht** auf **Interna** der Klasse zugreifen.
- ▶ Attribute a und i sind private → **kein Zugriff** von außen möglich.

```
public class ArrayStack<T>  
    implements Stack<T> {  
  
    private Object[] a;  
    private int i;  
  
    public ArrayStack() { // empty  
        a = new Object[1000];  
        i = -1;  
    }  
  
    public void push(T e) {  
        a[++i] = e;  
    }  
  
    public void pop() {  
        a[i] = null;  
        i--;  
    }  
  
    public T top() {  
        return (T)a[i];  
    }  
  
    public boolean isEmpty() {  
        return i== -1;  
    }  
}
```



Java-Stack: Implementierung

```
public class StackException  
    extends RuntimeException {  
  
    public StackException (String msg)  
    {  
        super(msg);  
    }  
  
    public void push(T e)  
        throws StackException {  
  
        if (i >= 999)  
            throw new  
                StackException("Full!");  
  
        a[++i] = e;  
    }  
  
    public void pop()  
        throws StackException {  
  
        if (isEmpty())  
            throw new  
                StackException("Empty!");  
  
        a[i] = null;  
        i--;  
    }
```

Die aktuelle Implementierung weist noch **Fehler** auf:

1. `pop()` funktioniert nicht bei **leerem Array**.
2. `push()` funktioniert nicht bei **vollem Array** (*1000 Elemente*).

Lösung: Exceptions

- Wir definieren eine Klasse `StackException`.
- Im Fehlerfall (*Überlauf / Unterlauf des Stacks*) signalisieren wir Fehler, indem wir Exceptions **werfen**.



Outline

1. Der Abstrakte Datentyp "Stack"

2. Stack: Implementierung

3. Dynamische Datenstrukturen

4. Verkettete Listen

5. Doppelt Verkettete Listen

Datenstrukturen: Motivation



```
public ArrayIntStack() {  
    a = new int[1000];  
    i = -1;  
}
```

Arrays als Datenstruktur

Wir haben bisher den Stack durch ein internes **Array** realisiert. **Was ist der Nachteil?**

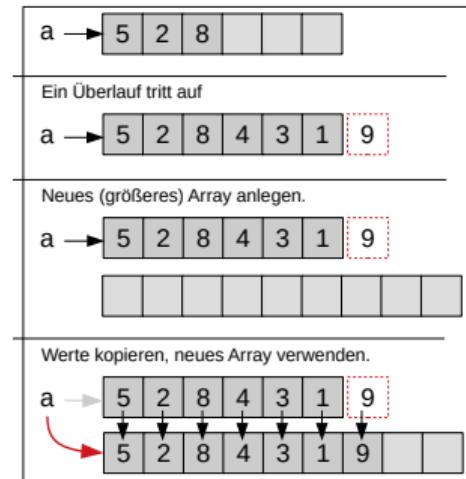
- ▶ Der Stack fasst nicht mehr als 1000 Elemente 😞
- ▶ Speicher wird ineffizient genutzt 😞
(bei vielen kleinen Stacks hoher Speicherverbrauch!).

(Teil-)Lösung: Dynamische Allokation

Bei einem Überlauf des Arrays ...

- ▶ ... wird ein um den Faktor α vergrößertes Array erstellt.
- ▶ ... und alle Elemente werden in das neue Array kopiert.

In Java: Typ `ArrayList` ($\alpha = 1.5$).





Datenstrukturen: Effizienz

Effizienz von Datenstrukturen

Wir beurteilen die Effizienz anhand folgender **Schlüsselfragen**:

- ▶ Was ist die Laufzeit, um ein neues Element **einzufügen**?
- ▶ Was ist die Laufzeit, um ein Element zu **suchen**?
- ▶ Was ist die Laufzeit, um ein Element zu **löschen**?
- ▶ Wieviel **Speicher** wird verbraucht?

Wir messen die Aufwände jeweils in **Abhängigkeit von n**, und im *Best, Worst und Average Case*.

Beispiel: Dynamisch allokierte Arrays

Sind dynamisch allokierte Arrays eine "gute" Datenstruktur?

Wir berechnen für Suchen, Einfügen und Löschen jeweils die
Worst-Case-Komplexität:

1. Suchen

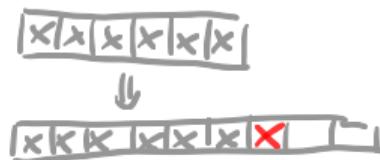


(Annahme: unsortiert)
 $\Rightarrow \Theta(n)$

2. Einfügen



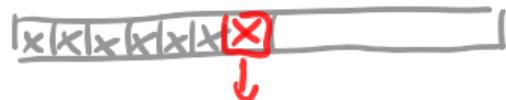
Im Normalfall: $\Theta(1)$
 (Wert an Stelle i schreiben)



Im Worst Case: $\Theta(n)$
 (Kopieren!!)

Beispiel: Dynamisch allokierte Arrays

3. Löschen:



$\text{pop}(): \Theta(1)$

innere Elemente löschen: $\Theta(n)$

Dynamische Datenstrukturen

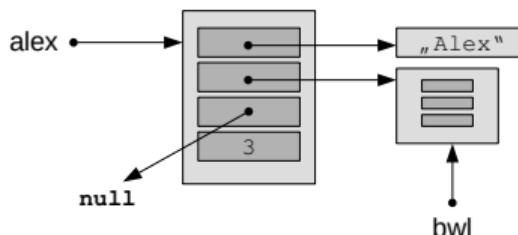
Array = Statische Datenstrukturen

- ▶ Fixe Größe, fixe Anordnung der Objekte im Speicher.

```
public class Student {
    String name;
    Studiengang studiengang;
    Bafoeg bafoeg;
    int semester;
    ...
}

Studiengang bwl = new Studiengang(...);

Student alex = new Student ("Alex", bwl,
                           null, 3);
```



Ab jetzt: Dynamische Datenstrukturen

- ▶ Struktur entsteht durch **Referenzierung (effizienter!).**
- ▶ Variable Größe.

Reminder: Referenzen

- ▶ Referenzen ≈ **Zeiger** auf Objekte (*sicher+getypt*).
- ▶ Wir verwenden in Java Referenzen, **nicht** die Objekte selbst!
- ▶ Jedes Objekt existiert (mindestens) solange eine Referenz auf es verweist.

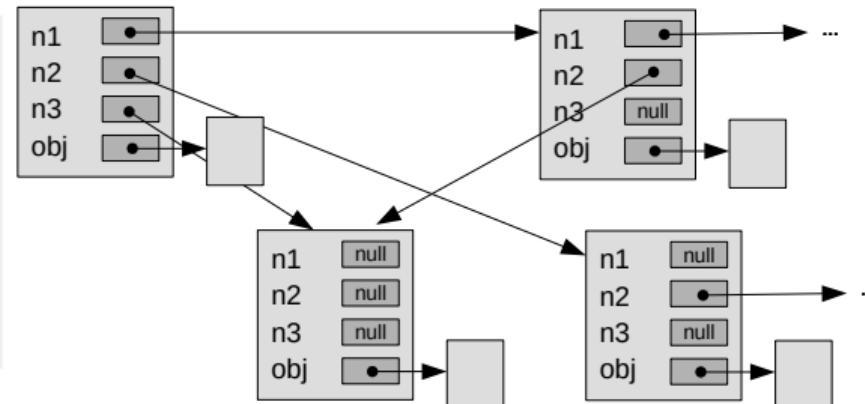
Dynamische Datenstrukturen: Collections



Schlüsselidee: Datenstruktur durch Referenzierung

- ▶ Objekte liegen nicht mehr in einem **kohärenten Speicherbereich** (siehe Array), sondern sind beliebig im Speicher **verteilt**.
- ▶ Objekte (oder “**Knoten**”) referenzieren andere Objekte desselben Typs.
- ▶ Durch diese Verkettung entsteht ein **Container-Datentyp** (engl. “*Collection*”)!

```
class Entry {  
    Entry n1;  
    Entry n2;  
    Entry n3;  
    Nutzdaten obj;  
    ...  
}
```



Dynamische Datenstrukturen: Collections

Collections bestimmter Typen?

Häufig speichern Collections Nutzdaten eines bestimmten Typs (*Kunden, Produkte, ...*).

```
// Polymorphie
class Entry {  
    Entry n1;  
    Entry n2;  
    Entry n3;  
    Object obj;  
    ...  
}
```

```
// Generics
class Entry<T> {  
    Entry<T> n1;  
    Entry<T> n2;  
    Entry<T> n3;  
    T obj;  
    ...  
}
```

```
// Polymorphie
// -> nicht typsicher :-(  
Entry e1 = new Entry();  
e1.obj = "Ein String";  
e1.obj = 17; // möglich!  
  
// Generics
// -> typsicher :-)  
Entry<String> e1 = new Entry<String>();  
e1.obj = "Ein String";  
e1.obj = 17; // nicht möglich!
```

Es gibt **zwei Optionen**:

1. **Polymorphie**: Nutzdaten `obj` vom Typ `Object`.
Verlust der Typsicherheit ☹
2. **Generics**: Ein zusätzlicher Parameter T
(sog. "Sortenparameter") gibt den Typ der Objekte an.



Outline

1. Der Abstrakte Datentyp "Stack"

2. Stack: Implementierung

3. Dynamische Datenstrukturen

4. Verkettete Listen

5. Doppelt Verkettete Listen

Einfachste dynamische Datenstruktur: Verkettete Listen



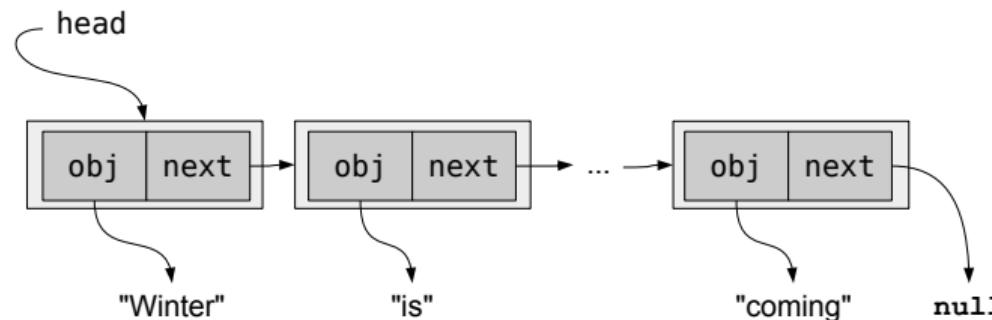
In **verketteten Liste** referenziert jeder **Knoten** auf maximal einen Nachfolger-Knoten.
Es ergibt sich eine Sequenz.

Attribute

Jeder Knoten besitzt **zwei Attribute**:

- ▶ obj: Referenz auf Nutzdaten (*im Bild unten: Strings*).
- ▶ next: Referenz auf nächsten Knoten. (*am Ende: null*).

Eine Referenz **head** auf den **ersten Knoten** dient zum “Einstieg”.



Verkettete Listen

Spezifikation

Die verkettete Listen soll folgende **Operationen** bieten:

- ▶ `addFirst()`/`addLast()`: Element vorne/hinten einfügen.
- ▶ `getFirst()`/`getLast()`: erstes/letztes Element erhalten.
- ▶ `removeFirst()`/`removeLast()`: erstes/letztes Element löschen.

Anmerkungen

Wir können z.B. **Stacks** mit verketteten Listen implementieren:

- ▶ `push()` → `addFirst()`
- ▶ `top()` → `getFirst()`
- ▶ `pop()` → `removeFirst()`

Verkettete Listen: Implementierung

```
class LinkedList<T> {  
    // information hiding:  
    // Interne Struktur nach  
    // außen nicht bekannt.  
    private Node head;  
  
    private class Node {  
        T obj;  
        Node next;  
  
        public Node(T obj) {  
            this.obj = obj;  
        }  
    }  
  
    public LinkedList() {  
        head = null;  
    }  
  
    public void addFirst(T obj);  
    public void addLast(T obj);  
    public T getFirst();  
    public T getLast();  
    public void removeFirst();  
    public void removeLast();  
}
```

Klassengerüst

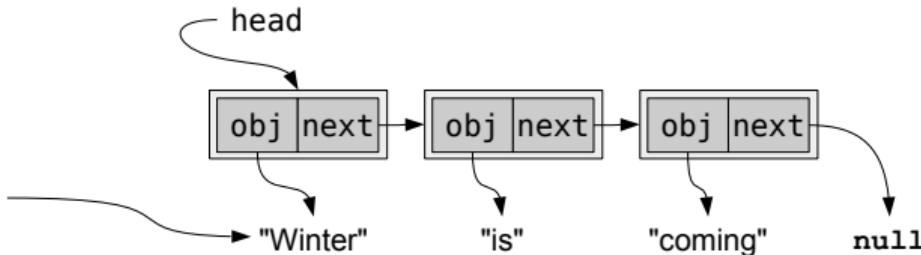
- ▶ Innere Klasse Node für die einzelnen Knoten der Liste.
- ▶ Referenz head auf ersten Knoten.

Information Hiding wird unterstützt

- ▶ Zugriff auf Details der Implementierung (*head, Node*) nicht möglich.
- ▶ **Existenz von Knoten unbekannt:**
Öffentliche Methoden geben keine Knoten nach außen, nur Nutzdaten!

Implementierung: getFirst()

Ergebnis
(Referenz auf
erstes Objekt)



```

public T getFirst()
    throws ListException {

    if (head == null) { // empty
        String msg = "List is empty.";
        throw new ListException(msg);
    }

    return head.obj;
}

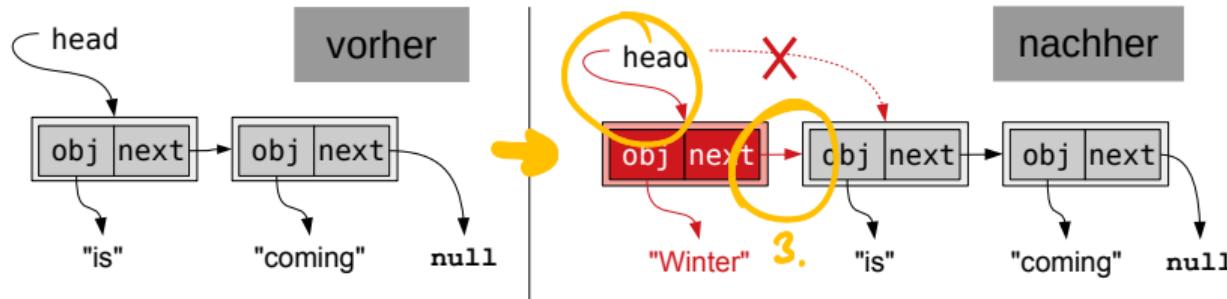
...
String s = list.getFirst();

```

Implementierung

1. Ausnahme falls Liste leer ist.
2. Ansonsten wird Referenz auf das **Nutzdatenobjekt** des heads zurückgegeben.

Implementierung: addFirst()



```

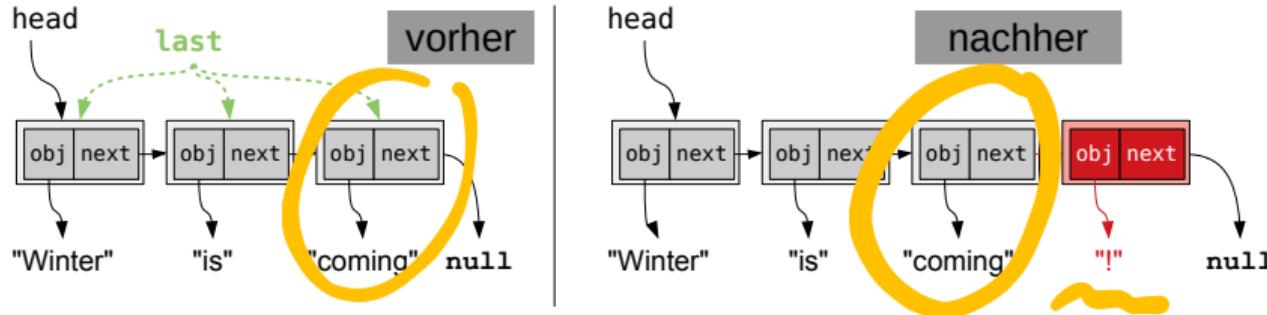
public void addFirst(T t) {
    Node n = new Node(t);      1.
    n.next = head;             3.
    head = n;                  2.
}
...
list.addFirst("Winter");

```

Implementierung

1. Neuer Knoten n (mit Referenz auf einzufügendes Objekt) wird erzeugt.
2. n wird am Anfang der Liste eingefügt (\rightarrow neuer Head).
3. Nächster Listenknoten ist der vorherige Head!

Implementierung: addLast()



```

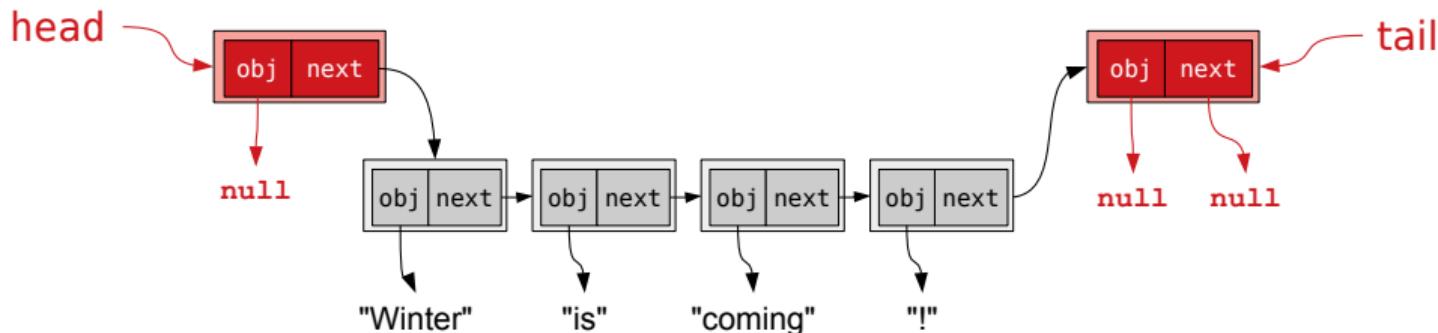
public void addLast(T t) {
    if (head == null) { } ①
        addFirst(t);
    Node last = head; } ②
    while (last.next != null) {
        last = last.next;
    Node n = new Node(t); } ③+④
    last.next = n;
}
...
list.addLast("!");

```

Implementierung

1. Sonderfall: Liste leer. !
2. Referenz `last` auf letzten Knoten bestimmten (*Liste durchlaufen*).
3. Neuen Noten erstellen, auf Knoten verweisen.
4. Nachfolger des ehemals letzten Knotens ist neuer Knoten.

Verbesserung der Implementierung: head und tail



Wir verwenden **Extra-Knoten** für **Anfang und Ende**:

- ▶ Anfangsknoten: head, Endknoten: tail.
- ▶ Beide Knoten besitzen **keine Nutzdaten**, sondern dienen ausschließlich zur **Navigation**.

Vorteile

- ▶ Vereinfacht Implementierung (*Bsp. removeLast()*).
- ▶ Einelementige Liste erfordert keine Spezialbehandlung mehr.



Outline

1. Der Abstrakte Datentyp "Stack"

2. Stack: Implementierung

3. Dynamische Datenstrukturen

4. Verkettete Listen

5. Doppelt Verkettete Listen

Verkettete Listen: Effizienz

Gegeben eine Liste mit n Elementen, was ist die (Worst Case)-Komplexität von...

- ▶ ... addFirst()? $\rightarrow \Theta(1)$ ☺ (*wir ändern nur zwei Referenzen*)
- ▶ ... addLast()? $\rightarrow \Theta(n)$ ☹ (*Durchlaufen der kompletten Liste*)

Lösung 1: Alle Referenzen “umdrehen”

- ▶ Nun hat addFirst() die Komplexität $\Theta(n)$ ☹

Lösung 2: Letztes Element (vor tail) speichern

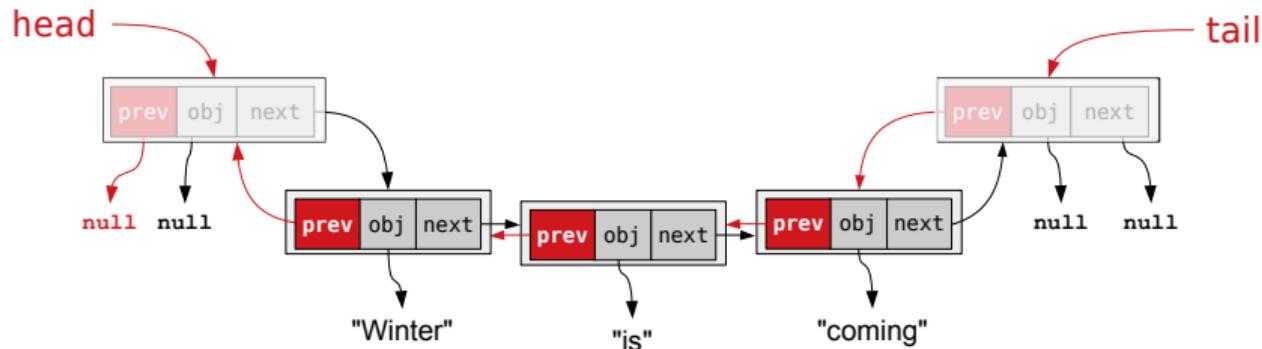
- ▶ Umständlich... was wenn das letzte Element entfernt wird? ☹

Lösung 3: Doppelt verkettete Liste

- ▶ Jedes Element verweist nicht nur auf seinen Nachfolger (next), sondern auch seinen Vorgänger (prev).



Doppelt Verkettete Listen



- ▶ obj: Referenz auf Nutzdaten
- ▶ next: Referenz auf nächstes Element (oder null, für tail)
- ▶ prev: Referenz auf vorheriges Element (oder null, für head)
- ▶ **Das letzte Element ist nun einfach tail.prev**
- Zugriff in **O(1)**.

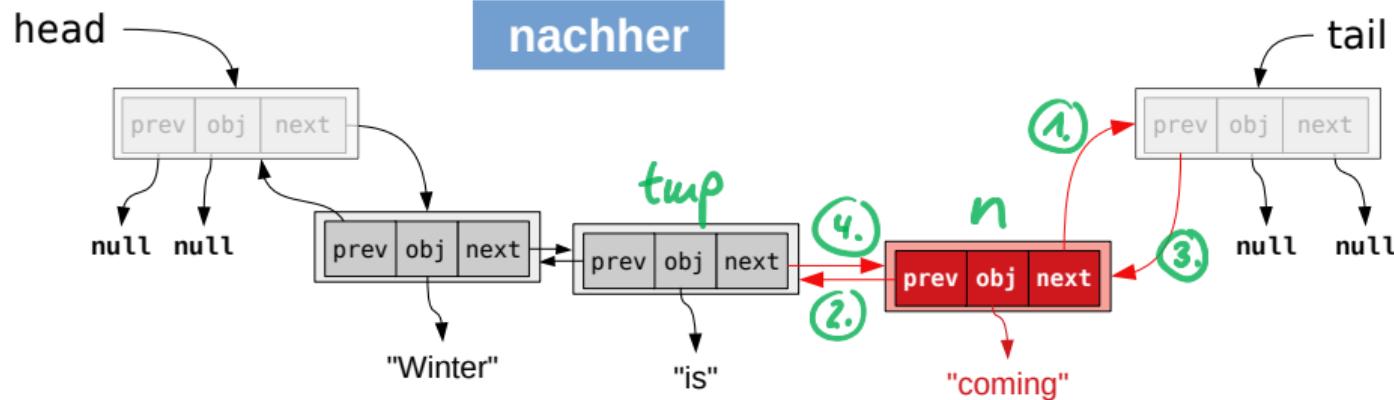
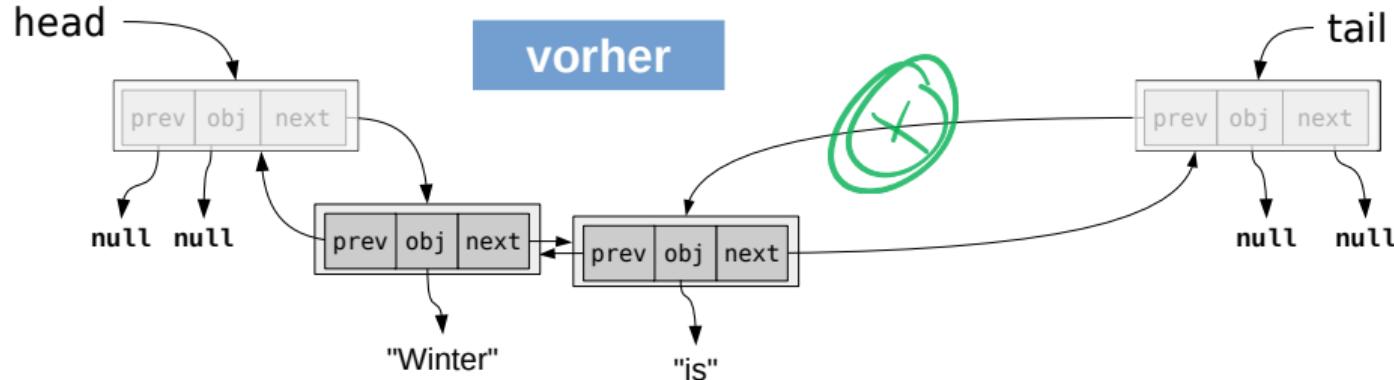


Doppelt Verkettete Listen: Implementierung

```
class DLinkedList<T> {  
  
    private Node head;  
    private Node tail;  
  
    private class Node {  
        T obj;  
        Node next;  
        Node prev;  
    }  
  
    public DLinkedList() {  
        head = new Node();  
        tail = new Node();  
        head.prev = null; head.next = tail;  
        tail.prev = head; tail.next = null;  
    }  
  
    public void addFirst(T obj);  
    public void addLast(T obj);  
    public T getFirst();  
    public T getLast();  
    public void removeFirst();  
    public void removeLast();  
}
```

- ▶ Innere private Klasse Node für die Knotenstruktur.
- ▶ Private Referenzen head und tail für den Einstieg.
- ▶ **Konstruktor** “Leere Liste”: Zwei Knoten head und tail, beide **verweisen aufeinander**.

Doppelt Verkettete Listen: addLast()



Doppelt Verkettete Listen: addLast()



```
public void addLast(T t) {  
    Node n = new Node(t);  
    Node temp = tail.prev;  
    n.next = tail; ①  
    n.prev = temp; ②  
    tail.prev = n; ③  
    temp.next = n;  
}
```

Sauberer Code! $\Theta(1)$ 😊

- Keine Schleife mehr
- Kein Spezialfall "Leere Liste" 😊

Doppelt Verkettete Listen: `length()` vs. `size()`

Wie implementieren wir `length()` (*#Elemente in der Liste*)?

```
int length() {
    int count;
    Node n = head.next;
    while (n != null) {
        count++;
        n = n.next;
    }
    return count;
}
```

```
// Komplexität? :-(
while (l.length() != 0) {
    l.pop();
}

// Komplexität? :-)
while (!l.isEmpty()) {
    l.pop();
}
```

Option 1: Elemente zählen

- ▶ Durchlaufe die Liste, inkrementiere Zähler für jedes Element.
- ▶ Nachteil? → Komplexität $\Theta(n)$.

Option 2: Separates Attribut 'length'

- ▶ Speichere Anzahl in Wert `length`, lese `length` aus.
- ▶ Vorteil? → Komplexität $\Theta(1)$.
- ▶ Nachteil? → `length`-Attribut muss gepflegt werden.

References I

