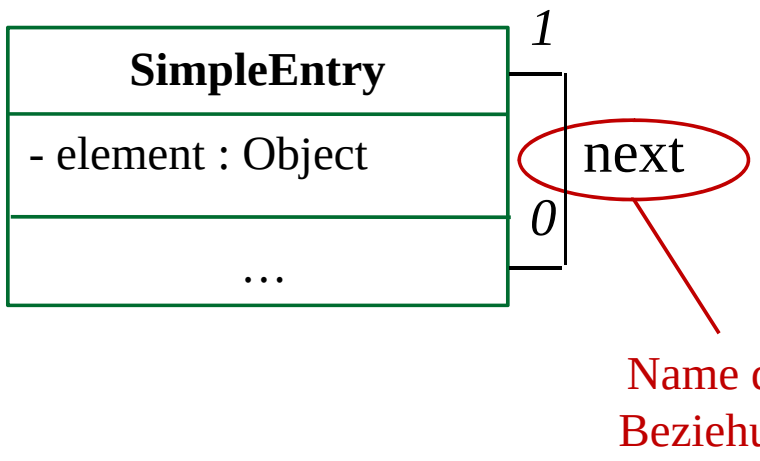
12.3 Ein Datenmodell für Listen

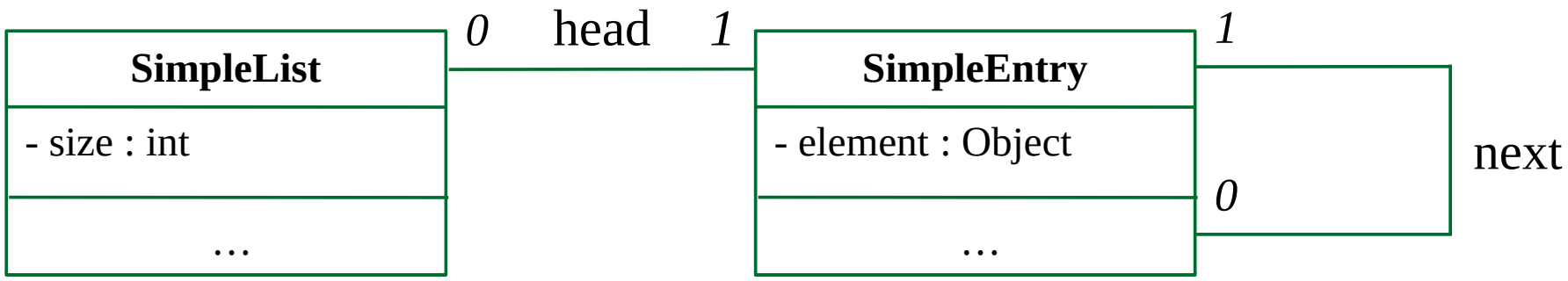
- Zweiter Versuch:
  - Wir modellieren ein Element der Liste zunächst als *eigenständiges Objekt* .
  - Dieses Objekt hält das gespeicherte Element.
  - Andererseits hält das Element-Objekt einen *Verweis* auf das nächste Element (Nachfolger).




```
public class SimpleEntry
{
  private Object element;
  private SimpleEntry next;

  public SimpleEntry(Object o, SimpleEntry next)
  {
    this.element = o;
    this.next = next;
  }

  public Object getElement()
  {
    return this.element;
  }

  public SimpleEntry getNext()
  {
    return this.next;
  }

  public void setNext(SimpleEntry next)
  {
    this.next = next;
  }
}
```

12.3 Ein Datenmodell für Listen

- Die eigentliche Liste benötigt nun nur noch einen Verweis auf das erste Element, d.h. die Klasse, die die Liste modelliert hat als Attribute ein Objekt vom Typ `SimpleEntry`.
  - Die Liste muss nur das *erste Element* kennen, über dessen Verweise zum nächsten Element (Attribut `next`) können alle Nachfolger erreicht werden.
  - In der leeren Liste ist das erste Element **`null`**.
- Zusätzlich speichert die Liste noch die Anzahl der Elemente in einem entsprechenden Attribut
- Modellierung der Klassen `SimpleList` und `SimpleEntry` in UML:



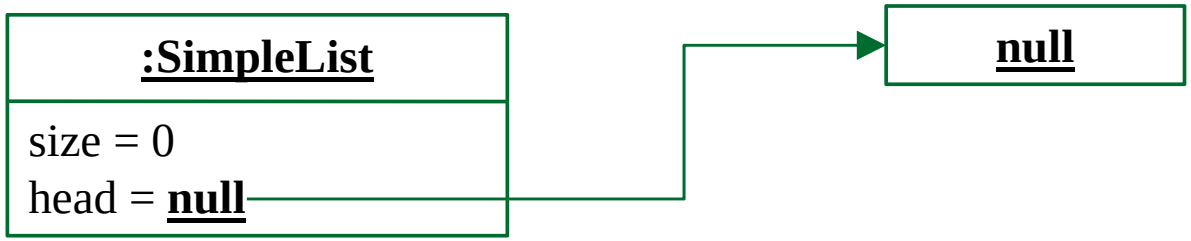
12.3 Ein Datenmodell für Listen


```
public class SimpleList
{
  private int size;

  private SimpleEntry head;

  public SimpleList()
  {
    this.size = 0;
    this.head = null;
  }

  public int length()
  {
    return this.size;
  }

  ...

}
```

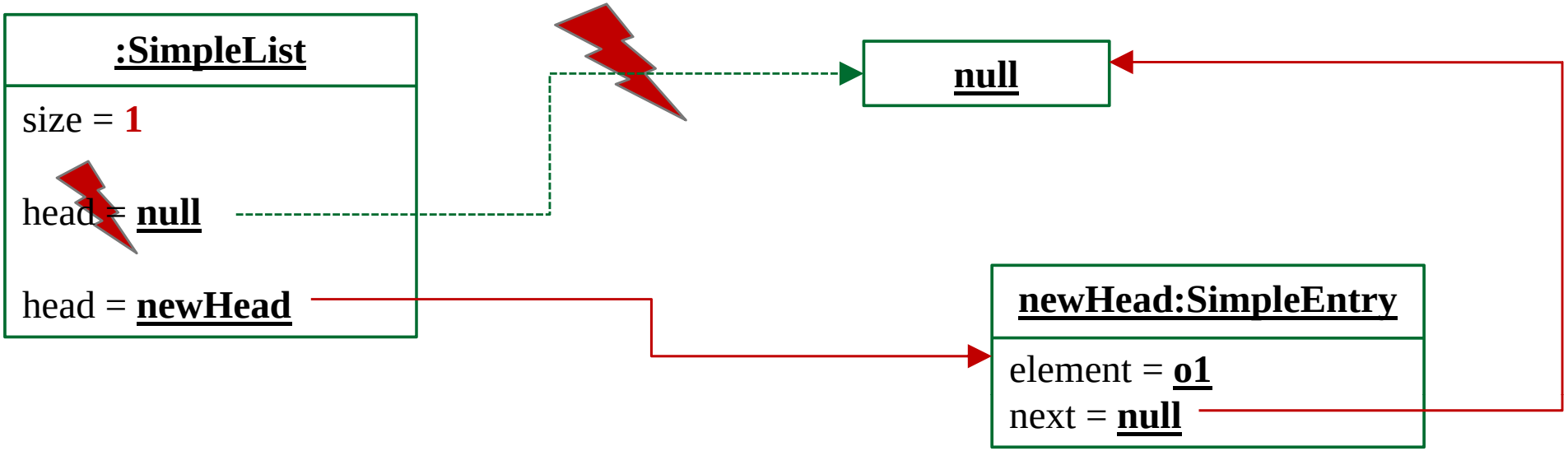
```
public class SimpleEntry
{
  private Object element;
  private SimpleEntry next;

  public SimpleEntry(Object o, SimpleEntry next)
  {
    this.element = o;
    this.next = next;
  }

  public Object getElement()
  {
    return this.element;
  }

  public SimpleEntry getNext()
  {
    return this.next;
  }

  public void setNext(SimpleEntry next)
  {
    this.next = next;
  }
}
```

12.3 Ein Datenmodell für Listen

- Veranschaulichung Der Konstruktor von `SimpleList` erzeugt eine leere Liste

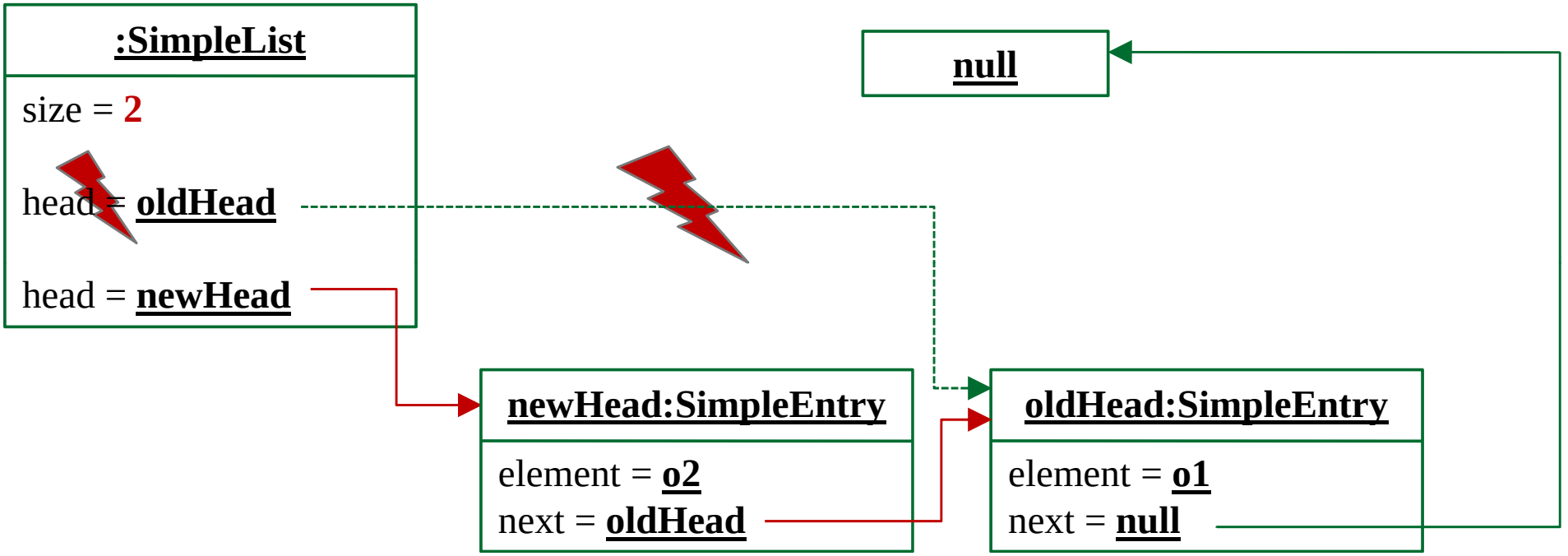


- Um ein neues Element hinzuzufügen, wird ein `SimpleEntry`-Objekt erzeugt und als neues erstes Element gesetzt. Dessen Nachfolger ist das alte erste Element. Die Länge der Liste erhöht sich um 1.



12.3 Ein Datenmodell für Listen

- Methode `prefix` bzw. `add`: Hinzufügen eines Elements (am Anfang)




```
...
public void add(Object o)
{
  SimpleEntry newHead = new SimpleEntry(o, this.head);
  this.head = newHead;
  this.size++;
}
...
```

Achtung, Kröger'sche Inkonsitenz: Methode sollte eigentlich `prefix` statt `add` heißen (siehe Folie 20)

12.3 Ein Datenmodell für Listen

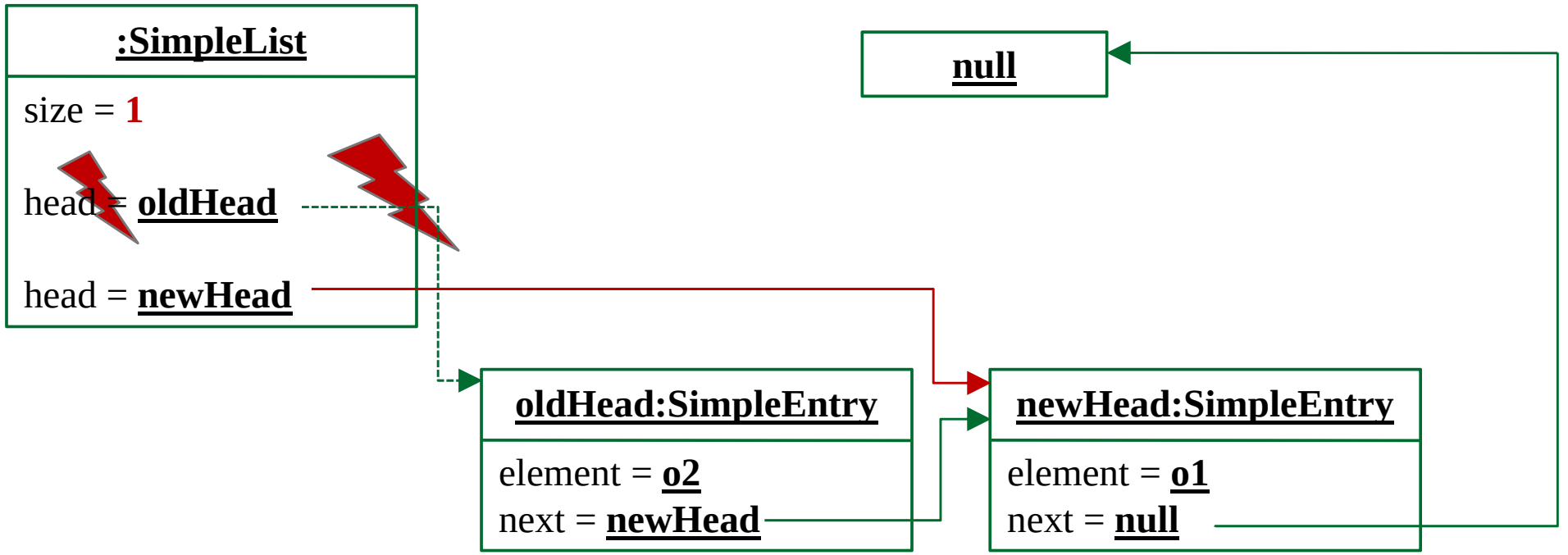
- Methode `first` bzw. `head` ergibt den Wert des ersten Elements der Liste

```
...

public Object head()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}

...
```

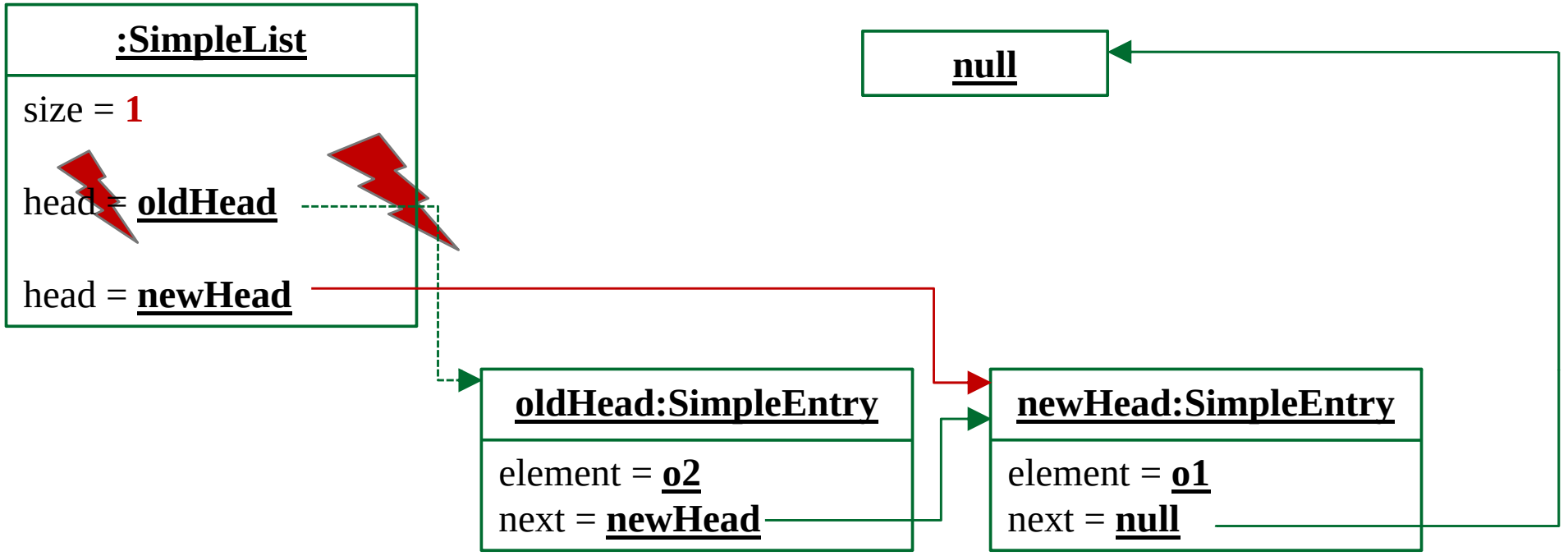
- Achtung bei leeren Listen!!!

12.3 Ein Datenmodell für Listen

- Wir betrachten nun noch eine Variante der Methode `rest`, bei der das erste Element der Liste gelöscht wird: `removeHead`
- Um das erste Element zu entfernen, muss der Head-Verweis auf den Nachfolger des ersten Elements zeigen und `size` erniedrigt werden.
- Danach gibt es keine Zugriffsmöglichkeit mehr für das erste Element!!!



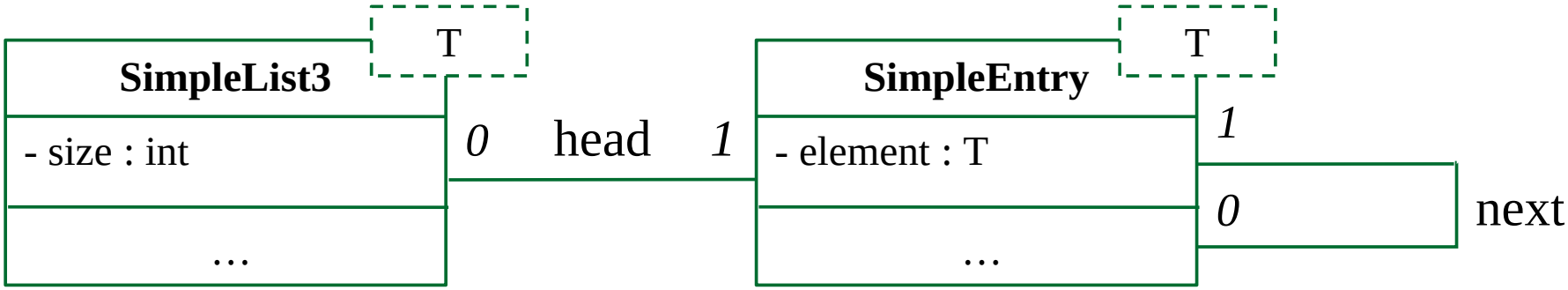
12.3 Ein Datenmodell für Listen




```
public void removeHead()
{
   if(this.head==null)
   {
      throw new NullPointerException("Empty List - no head element available.");
   }
   this.head = this.head.getNext();
   this.size--;
}
```

12.3 Ein Datenmodell für Listen

- Problem: Bisher können wir jedes beliebige Objekt in einer Liste ablegen
- Durch die Verwendung von `Object` als Typ des Listeneintrags sind in der Verwendung dem Auftreten von Typfehlern Tür und Tor geöffnet.
- Wir kennen schon eine Möglichkeit, die Liste typsicher zu machen: *Typisierung der Klasse*.



12.3 Ein Datenmodell für Listen

- Implementierung

```
public class SimpleList3<T>
{
  private int size;

  private SimpleEntry<T> head;

  public SimpleList3()
  {
    this.size = 0;
    this.head = null;
  }

  public int length()
  {
    return this.size;
  }
  ...
```

12.3 Ein Datenmodell für Listen


```
...
public void add(T o)
{
  SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
  this.head = newHead;
  this.size++;
}

public T head()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  return this.head.getElement();
}

public void removeHead()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  this.head = this.head.getNext();
  this.size--;
}
...
```

12.3 Ein Datenmodell für Listen


```
public class SimpleEntry<T> {
      private T element;

      private SimpleEntry<T> next;

      public SimpleEntry(T obj, SimpleEntry<T> next) {
               this.element = obj;
               this.next = next;
      }

      public T getElement() {
               return this.element;
      }

      public SimpleEntry<T> getNext() {
               return this.next;
      }

      public void setNext(SimpleEntry<T> next) {
               this.next = next;
      }
}
```

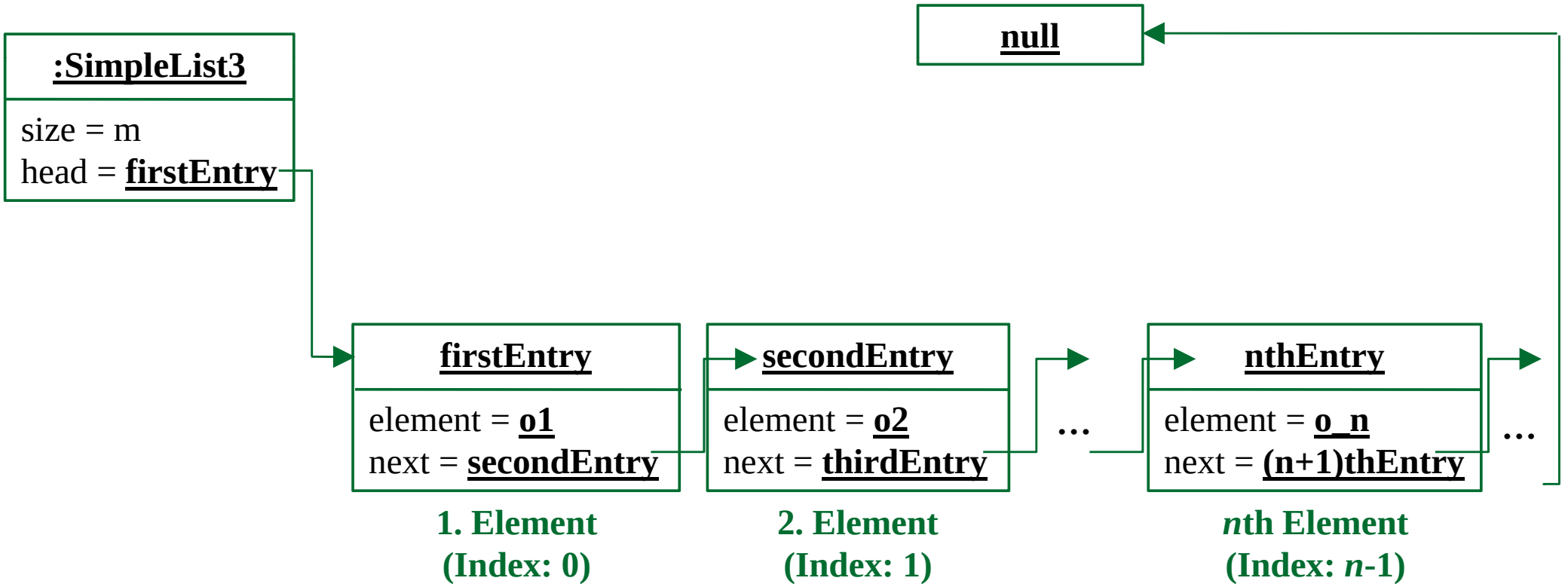
12.3 Ein Datenmodell für Listen

- Vergleich Liste vs. Array:
  - Dynamisches Wachstum durch Einfügen eins Elements am Anfang der Liste?
  - Dynamisches Schrumpfen durch Löschen des ersten Elements?
  - Zugriff auf das erste Element?
  - Zugriff auf das *n*-te Element?
  - Einfügen/Löschen des *n*-ten Elements?
  - Suche nach einem Element?
  - Zwei Listen/Arrays verschmelzen?

12.3 Ein Datenmodell für Listen

- Beispiel: Zugriff auf das *n*-te Element

  Um das *n*-te Element einer Liste zu bekommen, müssen die ersten *n* – 1 Elemente durchlaufen werden (wir verwenden die selbe Indexmenge wie bei Arrays)



12.3 Ein Datenmodell für Listen


```
...
public T get(int index)
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List.");
  }
  if(index>=this.length())
  {
    throw new IllegalArgumentException(index+" exceeds length of list.");
  }
  SimpleEntry<T> currentEntry = this.head;
  while(index>0)
  {
    currentEntry = currentEntry.getNext();
    index--;
  }
  return currentEntry.getElement();
}
...
```

12.3 Ein Datenmodell für Listen

Analog:

- Um das *n*-te Element aus einer Liste zu entfernen, müssen ebenfalls die ersten *n* – 1 Elemente durchlaufen werden.
- Dann muss der Verweis des (*n* – 1)-ten Elements auf den neuen Nachfolger "umgebogen" werden

- Um ein Element an einer bestimmten Stelle *n* einzufügen, müssen wiederum die ersten *n* – 1 Elemente durchlaufen werden.
- Dann muss der Verweis des (*n* – 1)-ten Elements auf den neuen Nachfolger "umgebogen" werden.
- Zuvor brauchen wir aber den alten Verweis, weil der neue Nachfolger des (*n* – 1)-ten Elements als Nachfolger den alten Nachfolger des (n – 1)-ten Elements haben muss.

12.3 Ein Datenmodell für Listen

Suche nach einem spezifischen Element o

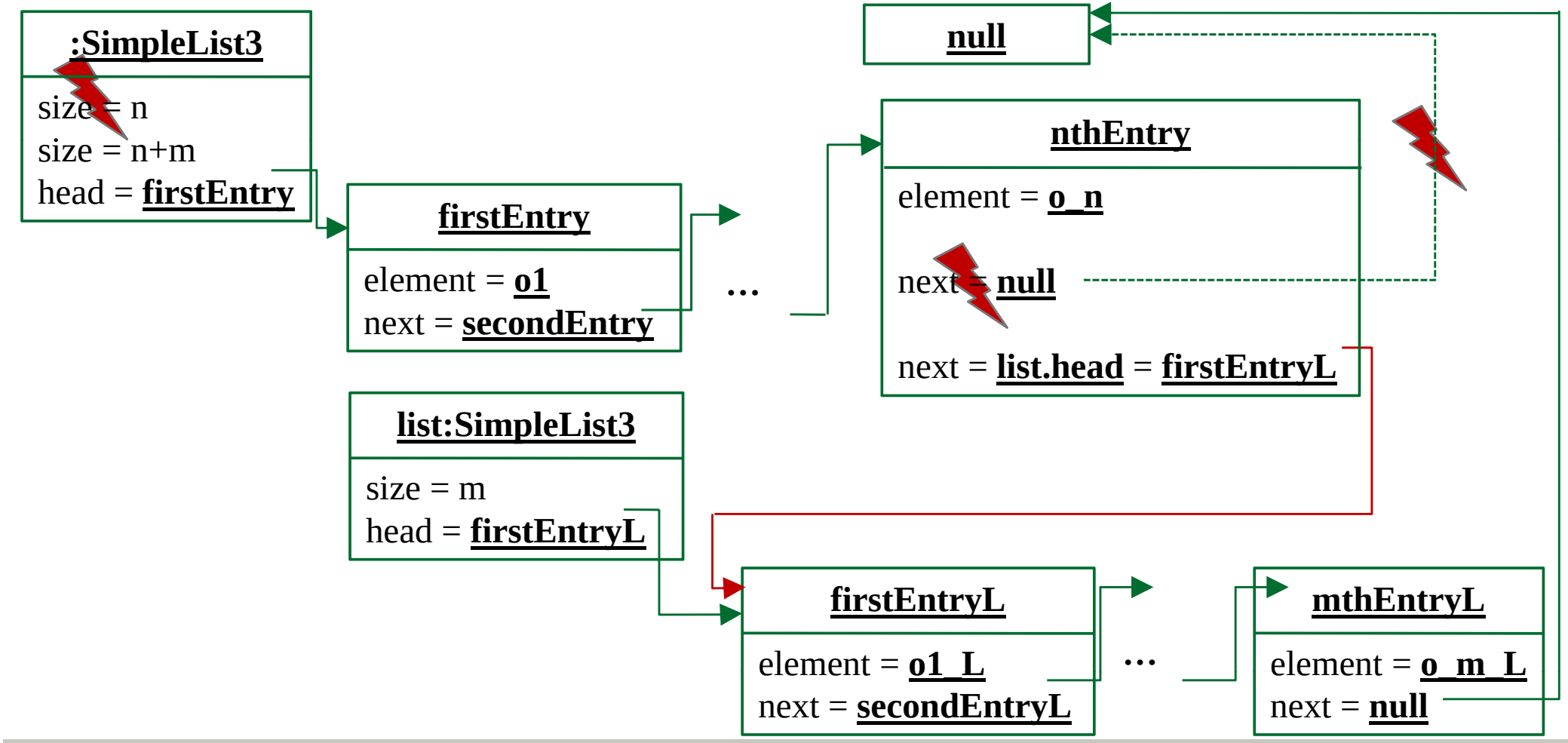
```
...
public boolean contains(T o)
{
  SimpleEntry<T> currentEntry = this.head;
  while(currentEntry!=null && !currentEntry.getElement().equals(o))
  {
    currentEntry = currentEntry.getNext();
  }
  return currentEntry!=null;
}
...
```

- Achtung: T sollte `equals` entsprechend überschreiben, um auf Gleichheit statt Identität zu prüfen

12.3 Ein Datenmodell für Listen

„Konkatenation“ zweier Listen: Methode append

- Das letzte Element der einen Liste soll nicht mehr auf **`null`** verweisen, sondern auf das erste Element der anderen Liste



12.3 Ein Datenmodell für Listen

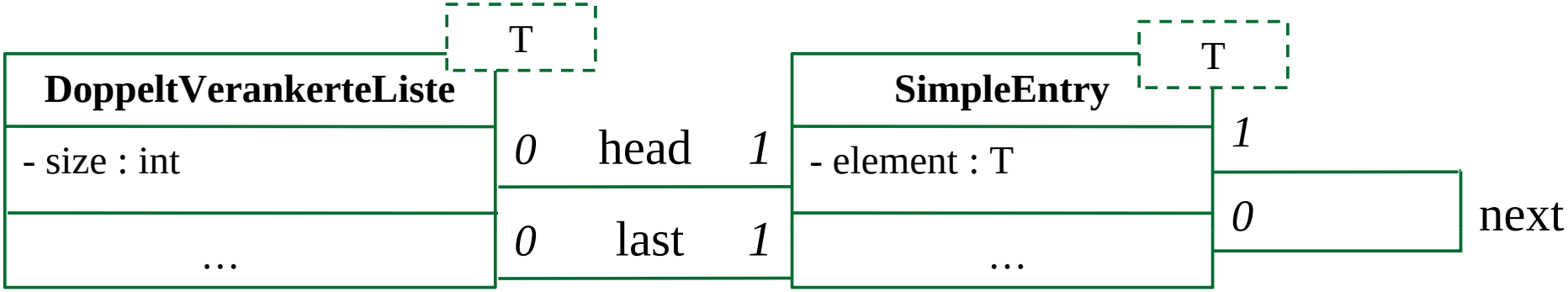

```
public void append(SimpleList3<T> list)
{
  SimpleEntry<T> last = this.head;
  while(last.getNext()!=null)
  {
    last = last.getNext();
  }
  last.setNext(list.head);
  this.size += list.length();
}
```

- Komplexität ist $O(n)$, da man die erste Liste ganz durchwandern muss
- Kann man die Liste so implementieren, dass sich der Zeitbedarf für append auf $O(1)$ reduziert?

Übersicht

12.4 Varianten

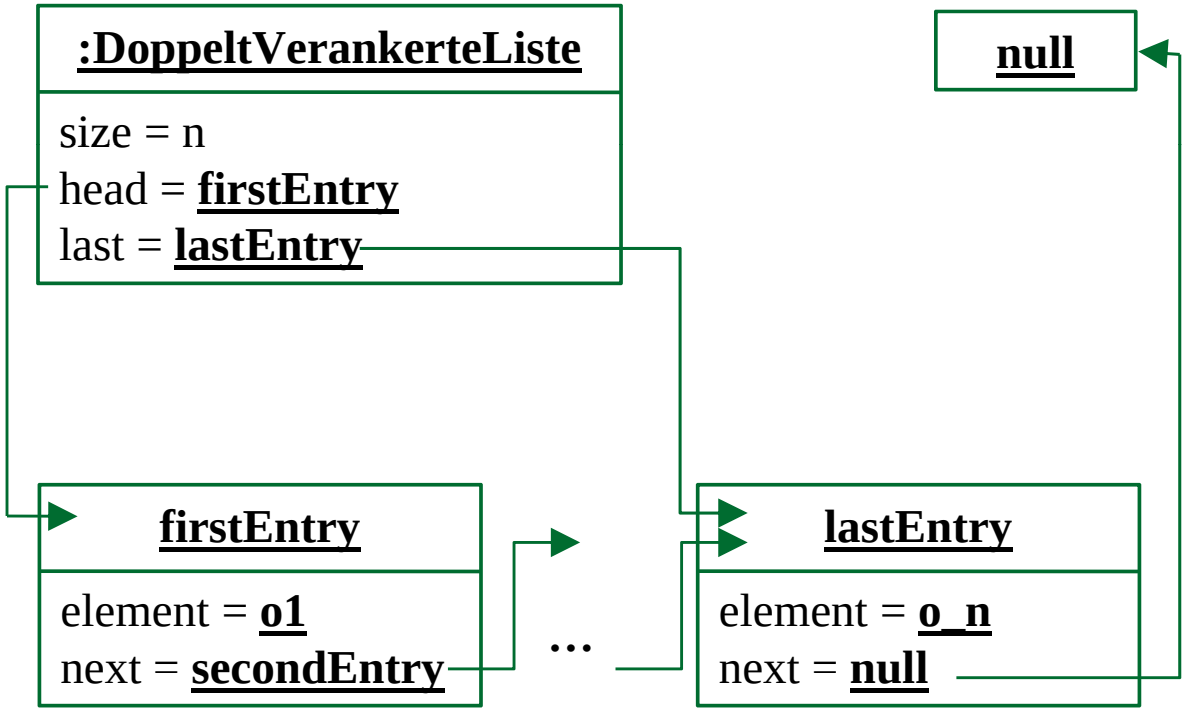
- Lösung: eine *doppelt-verankerte Liste* hält den Verweis auf das erste *und* letzte Element jeweils als Attribut




```
public class DoppeltVerankerteListe<T>
{
  private int size;

  private SimpleEntry<T> head;

  private SimpleEntry<T> last;

  public DoppeltVerankerteListe()
  {
    this.size = 0;
    this.head = null;
    this.last = null;
  }
  ...
```

12.4 Varianten

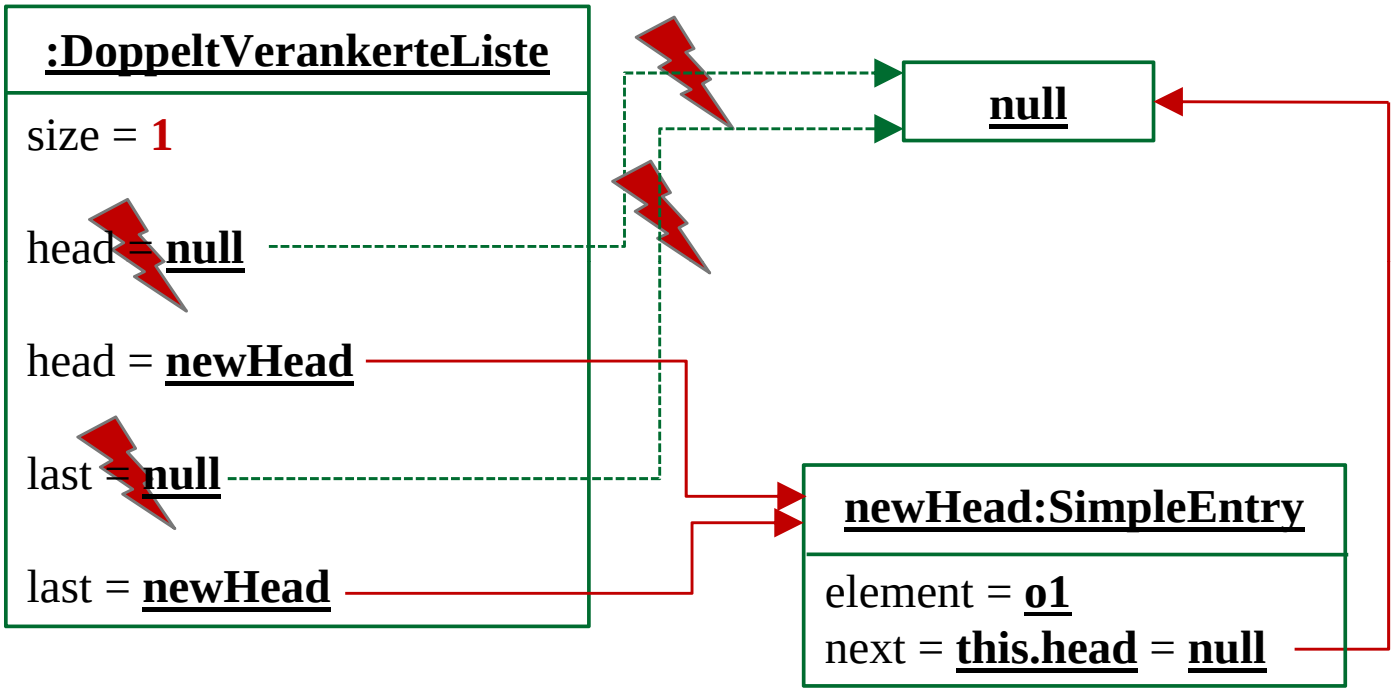
Allgemeines Schema:



Nach Instanziierung:

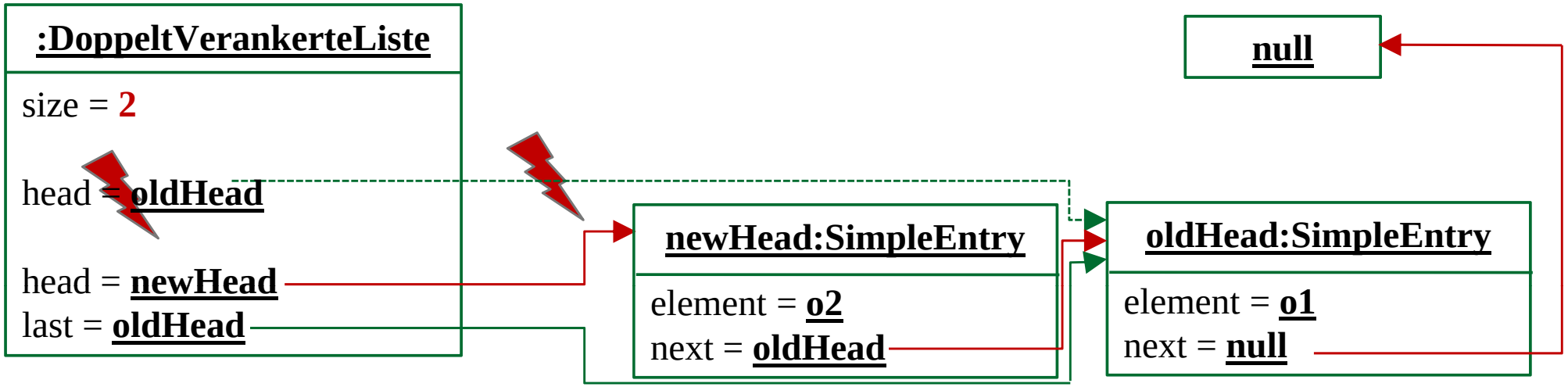
12.4 Varianten

Erstes Einfügen:



12.4 Varianten

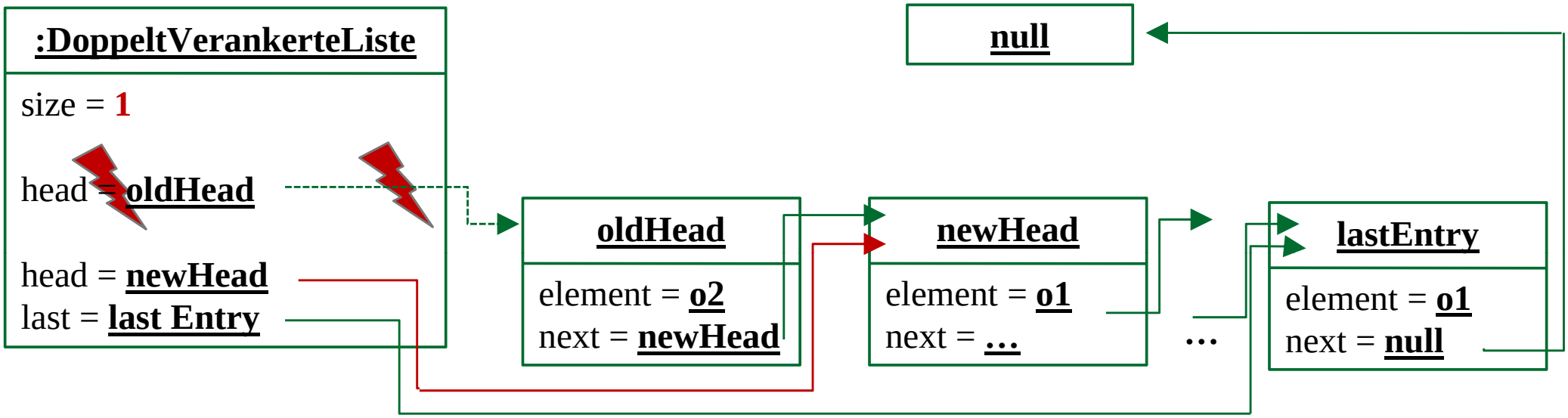
Zweites Einfügen:




```
public void add(T o)
{
  SimpleEntry<T> newHead = new SimpleEntry<T>(o, this.head);
  this.head = newHead;
  if(this.last==null)
  {
    this.last = newHead;
  }
  this.size++;
}
```

12.4 Varianten

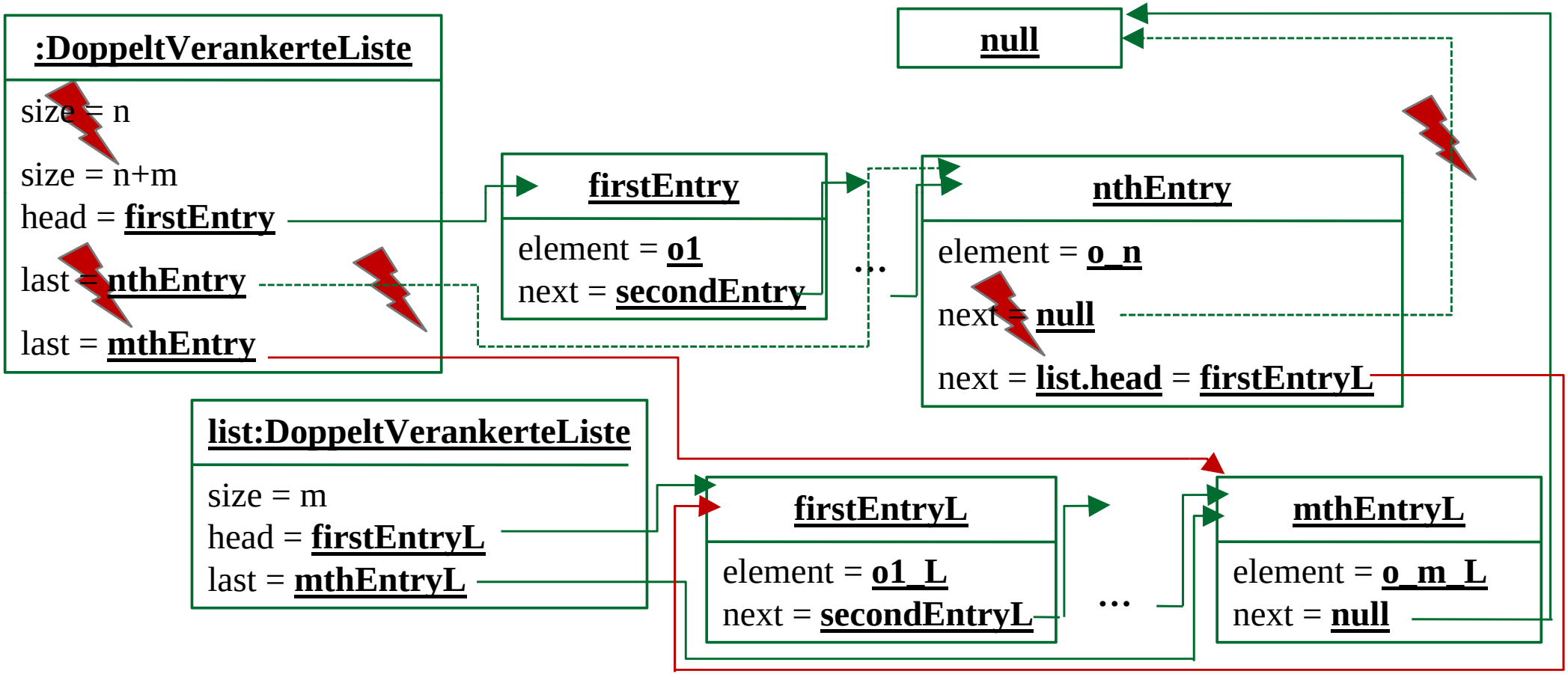
Erstes Element entfernen:




```
public void removeHead()
{
  if(this.head==null)
  {
    throw new NullPointerException("Empty List - no head element available.");
  }
  this.head = this.head.getNext();
  if(this.head==null)
  {
    this.last = null;
  }
  this.size--;
}
```

12.4 Varianten

Zwei Listen zusammenfügen:



12.4 Varianten


```
public void append(DoppeltVerankerteListe<T> list)
{
  this.last.setNext(list.head);
  this.last = list.last;
  this.size += list.length();
}
```

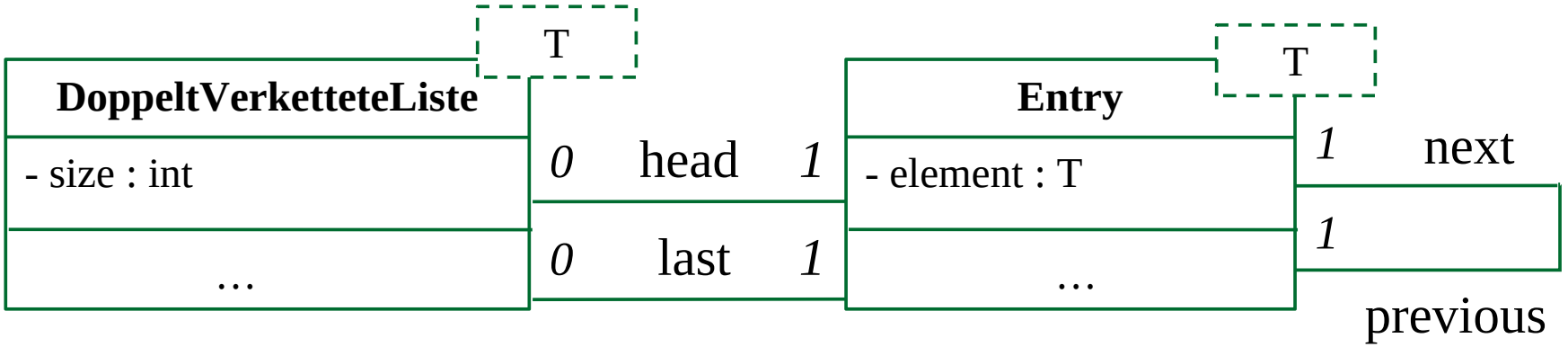
- Damit haben wir gleichzeitig eine effiziente Möglichkeit gewonnen, ein einziges Element am Ende der Liste einzufügen.

```
public void append(T o)
{
  SimpleEntry<T> newLast = new SimpleEntry<T>(o, null);
  this.last.setNext(newLast);
  this.last = newLast;
  this.size++;
}
```

- Ist auch das Entfernen am Ende der Liste möglich?
  - Der Zeiger für das letzte Element muss auf das vorletzte Element "umgebogen" werden wofür wir wieder von vorne die Liste durchlaufen müssen

12.4 Varianten

- Lösung: eine *doppelt-verkettete Liste* speichert für jeden Eintrag auch einen Verweis auf dessen Vorgänger.
- Dadurch kann man die Liste auch von hinten nach vorne durchlaufen (und damit Entfernen des letzten Elements in $O(1)$ möglich).



12.4 Varianten

- Eine doppelt-verkettete Liste ist die flexibelste Implementierung:
  - Es können Elemente am Anfang und Ende der Liste effizient angefügt und entfernt werden
  - Dies ist die allgemeinste Umsetzung von Folgen, die wir in Kap 3 alternativ durch
    1. $() \in M^*$
    2. Ist $x \in M^*$ und $a \in M$, dann ist *postfix* $(x, a) \in M^*$.

    bzw. durch
    1. $() \in M^*$
    2. Ist $a \in M$ und $x \in M^*$, dann ist *prefix* $(a, x) \in M^*$.

    definiert haben

- Standard-Implementierungen einer verketteten Liste sind daher oft doppelt-verkettete Listen (z.B. `java.util.LinkedList`).

12.4 Varianten

- Neben der allgemeinen Liste, wie auf der vorherigen Folie beschrieben gibt es oft spezielle Varianten für verschiedene Anwendungen, die die Möglichkeiten für das Einfügen und für den Zugriff auf Elemente der Liste einschränken
- Zwei wichtige Varianten sind:
  - *Keller* (*Stack*) als LIFO-Datenstruktur mit den Methoden
    - `void push(T o)` legt Objekt `o` oben auf dem Stapel ab.
    - `T pop()` entfernt das oberste Element (vom Typ `T`) und gibt es zurück.
  - (*Warte-*)*Schlange* (*Queue*) als FIFO-Datenstruktur mit den Methoden
    - `void put(T o)` fügt das Objekt `o` hinten an die Schlange an.
    - `T get()` entfernt das vorderste Element (vom Typ `T`) und gibt es zurück.

12.4 Varianten

Diskussion

- Den Vorteil der flexiblen Länge haben wir in den bisherigen Implementierungen dadurch erkauft, dass der (wahlfreie) Zugriff auf das n-te Element eine Zeitkomplexität von $O(n)$ hat.
- Andere Implementierungen verwirklichen die flexible Länge durch ein internes Array (meist initial mit einer speziellen Größe) und ermöglichen dadurch wieder wahlfreien Zugriff in $O(1)$.
- Wird das interne Array zu kurz, wird es durch ein längeres ersetzt und der Inhalt des alten Arrays in das neue kopiert (siehe erster Versuch).
- Solche array-basierten Listen-Implementierungen in der Java API sind `java.util.ArrayList` und `java.util.Vector`.
- Der Vorteil durch zeiteffizienten wahlfreien Zugriff ($O(1)$) wird erkauft durch höheren Speicherplatzbedarf und gelegentlichen Kopieraufwand beim Wachsen der Liste was wiederum in $O(n)$ liegt.