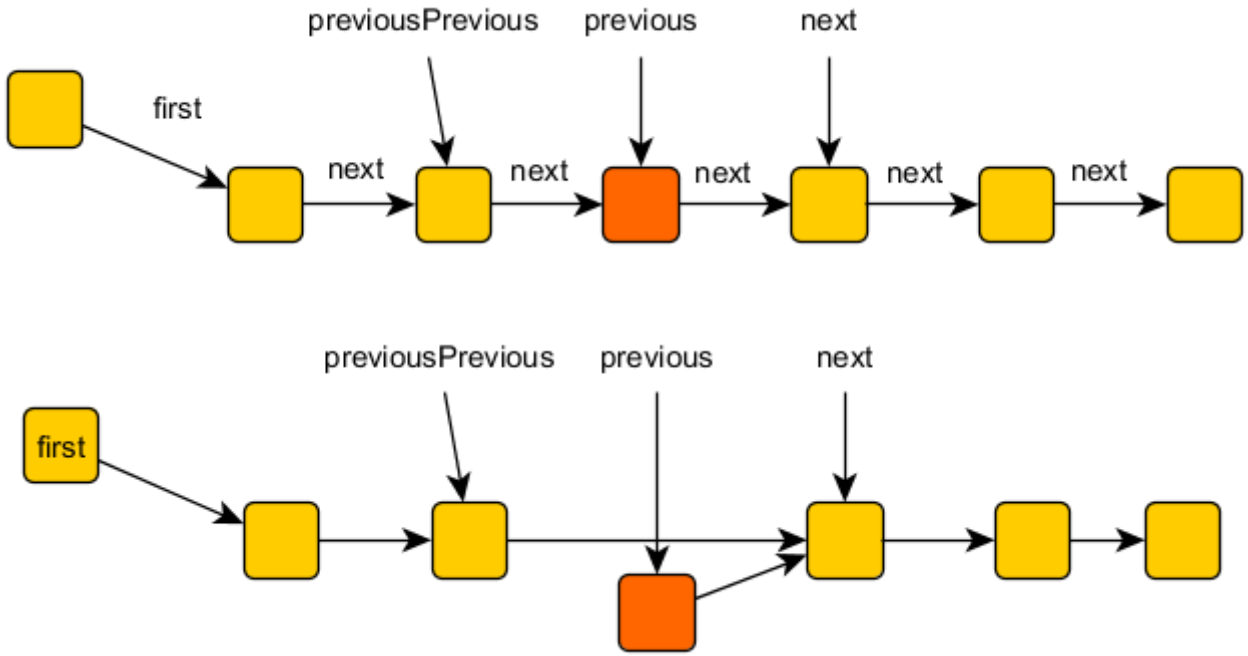
Feedback: Iteratoren
Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

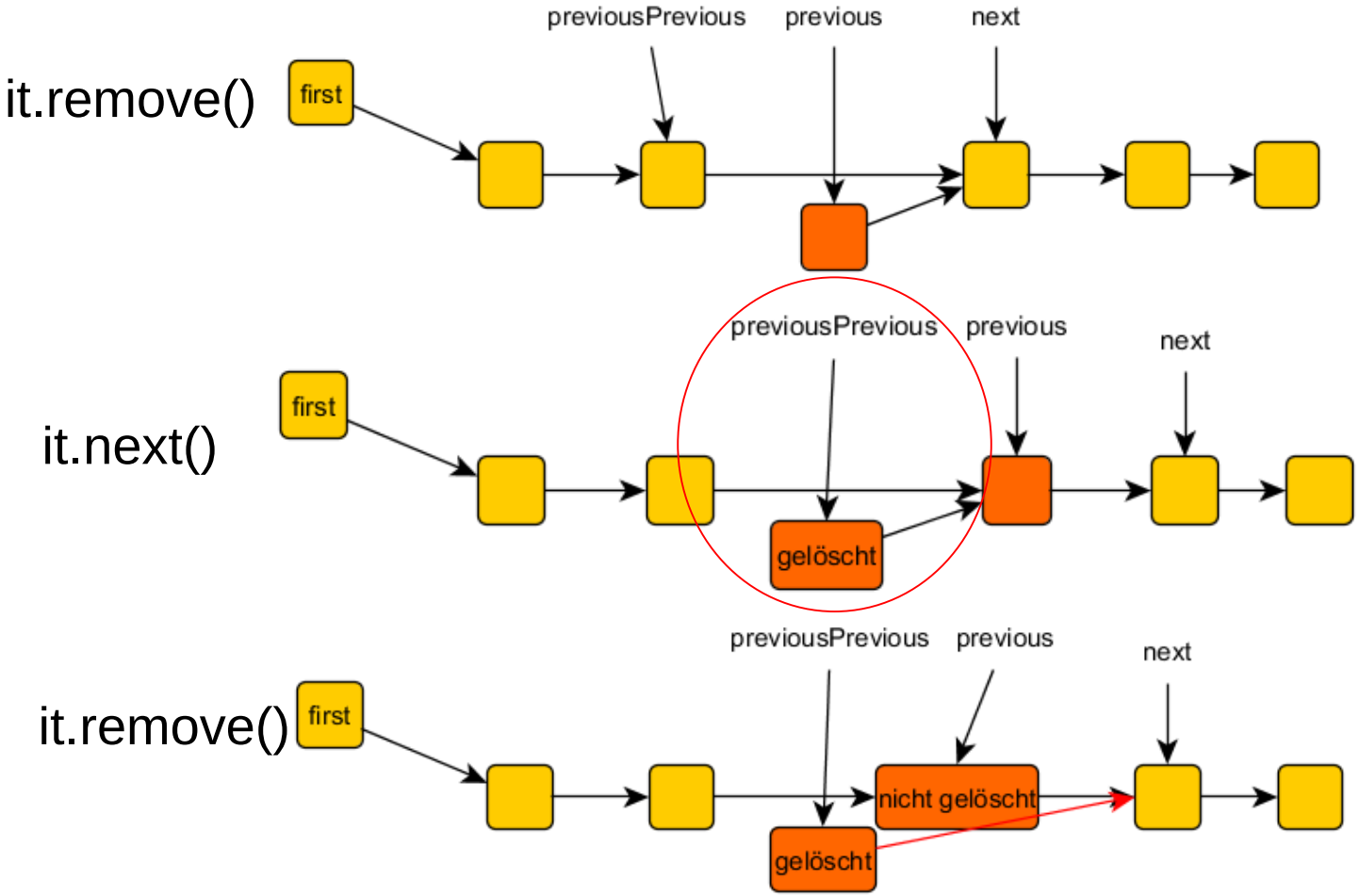
Remove auf Iterator (einfach verkettete Liste)

- Löscht das zuletzt zurückgegebene Element



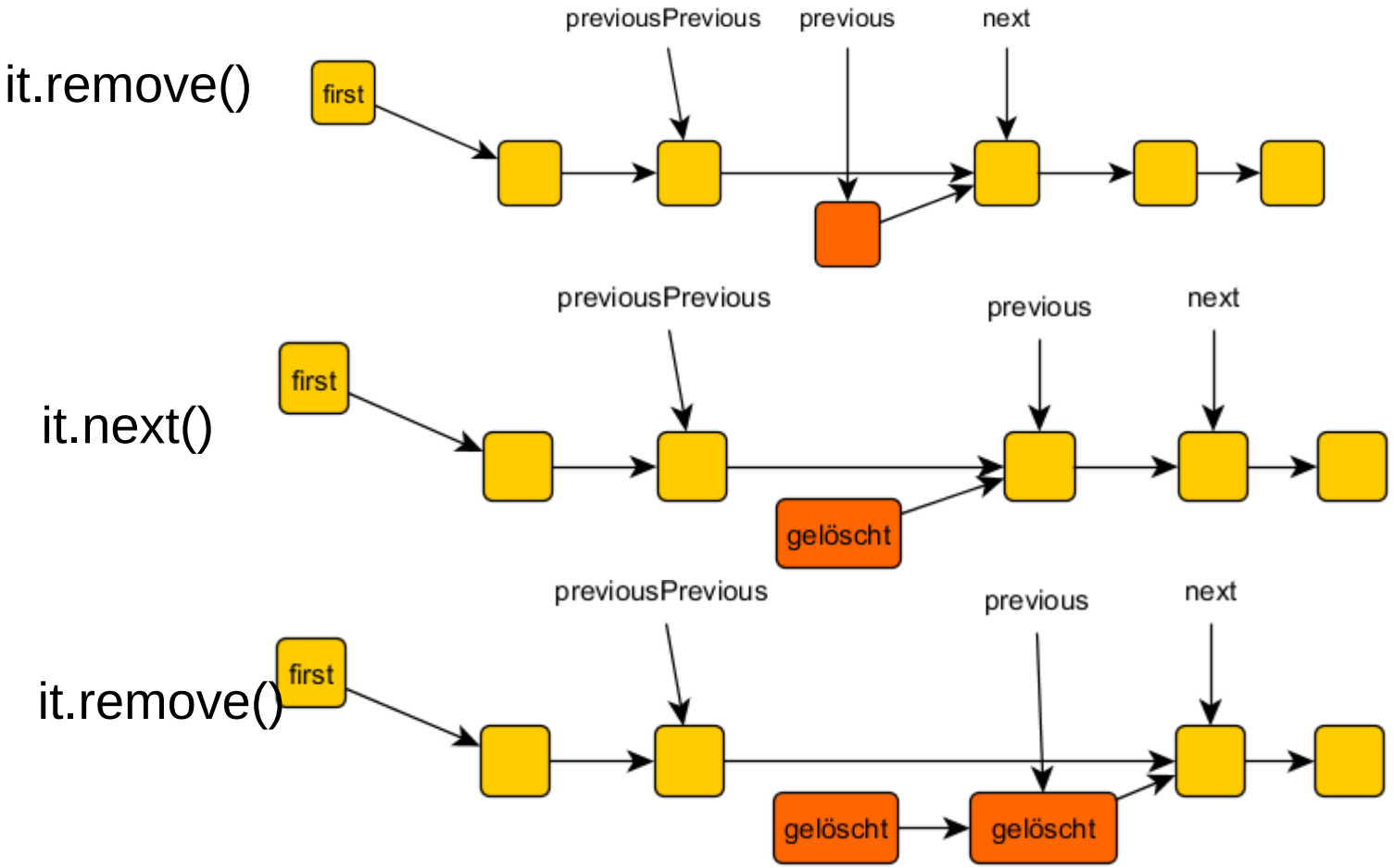
Remove auf Iterator (einfach verkettete Liste): Falscher Zeiger nach Löschen

- PreviousPrevious wandert zu Previous! (Falsch, weil nicht mehr in Liste)



Remove auf Iterator (einfach verkettete Liste): Korrekte Zeiger nach Löschen

- PreviousPrevious bleibt nach Löschen stehen!



Iterator vs. ListIterator

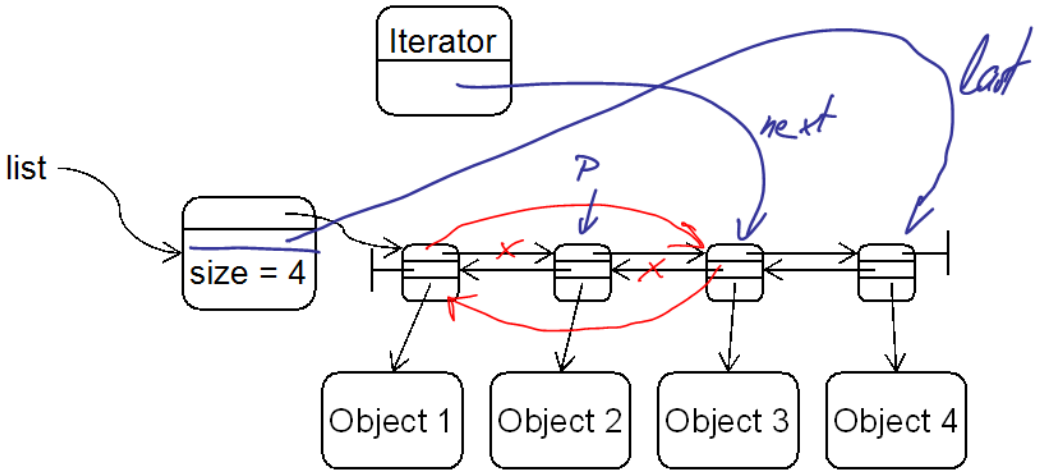
- Iterator
  - `boolean hasNext()`
  - `E next()`
  - `void remove()`

- ListIterator hat zusätzlich:
  - `boolean hasPrevious()`
  - `E previous();`
  - `int previousIndex();`
  - `void set(E e)`
  - `void add(E e)`

Zu beachten bei der Iterator-Implementation

- **First- und Last-Zeiger** auf der Liste müssen ggf. angepasst werden beim:
  - Entfernen auf dem Iterator
  - Hinzufügen über den ListIterator
- **ModCount** wird nur erhöht

3.6 / 3.7 Doppelt verkettete Liste (Script)



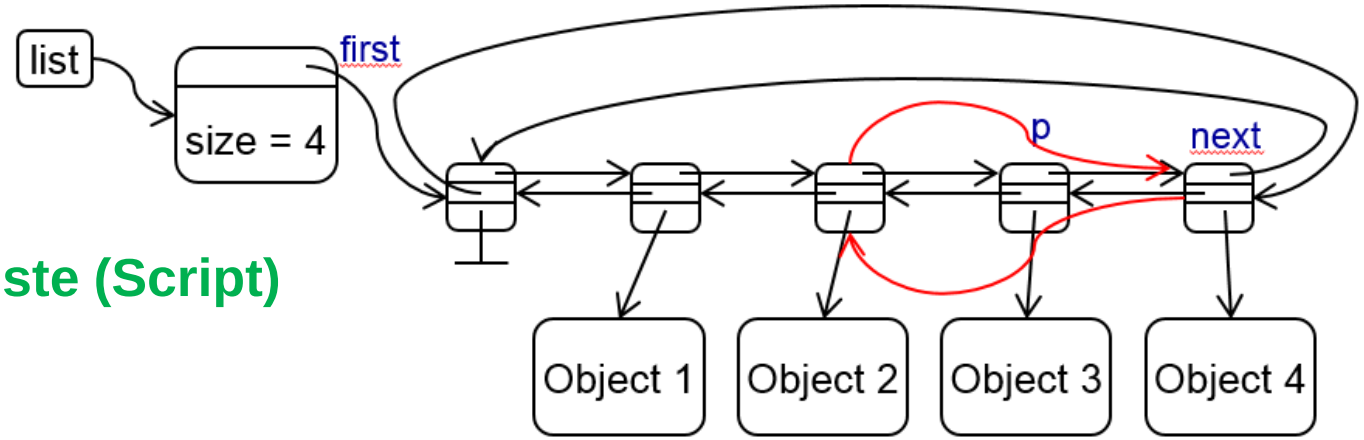

```
class MyIterator<E> implements Iterator<E> {
  private boolean mayRemove = false;
  private Node<E> next = first;

  public boolean hasNext() { ... }
  public E next() { ... mayRemove = true; ... }
  public void remove() {
    if (!mayRemove) throw new IllegalStateException();
    Node<E> p = next != null ? next.prev : last;
    if(p.prev != null) p.prev.next = p.next;
    else first = p.next;
    if (p.next != null) p.next.prev = p.prev;
    else last = p.prev;
    mayRemove = false; size--;
  }
}
```

// nicht 1. Element

// nicht letztes Element

```
++iteratorModCount;
++listModCount;
```



3.8 Doppelt verkettete Ringliste (Script)

```
private boolean next = first.next
public boolean hasNext() {
    return next != first;

}
public void remove() {
    if (!mayRemove) throw new IllegalStateException;
    Node<E> p = next.prev;
    p.next.prev = p.prev;
    p.prev.next = p.next;
    mayRemove = false; size--; ++iteratorModCount;
                               ++listModCount;
}
```

Lernziele Listen / Iteratoren

Listen:

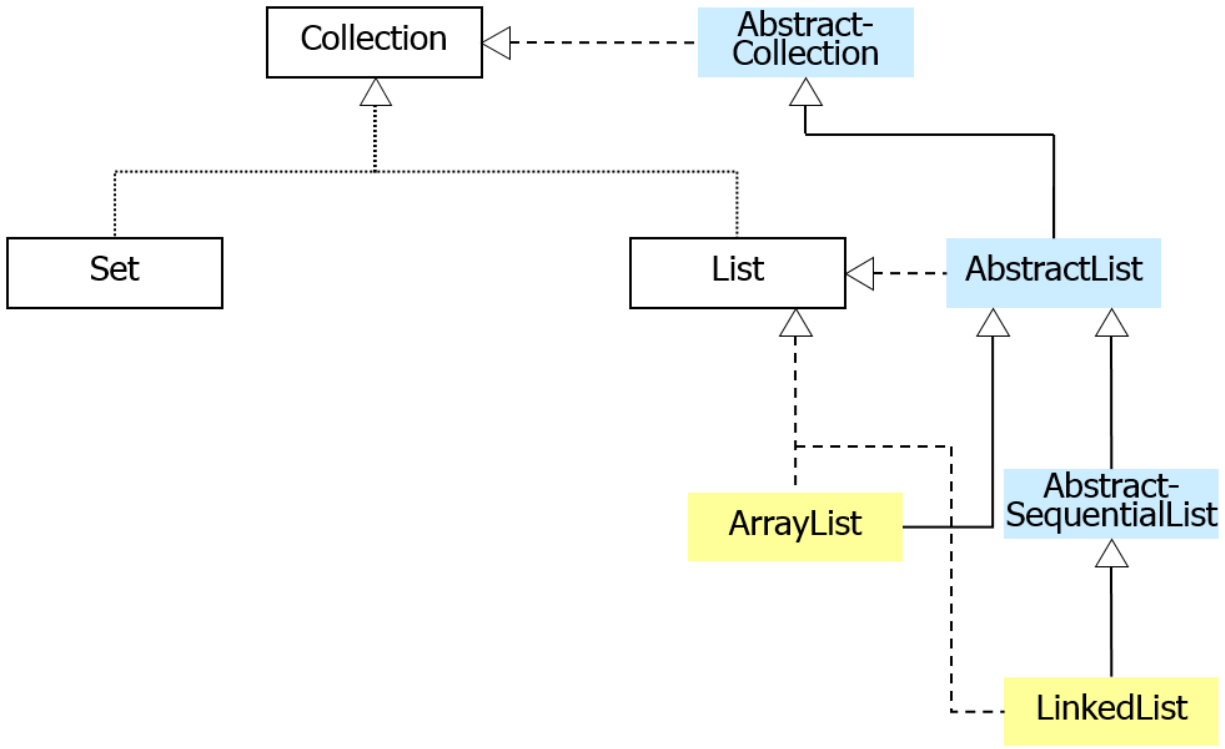
- Sie können verlinkte Listen in Java implementieren.
- Sie können für einige Implementierungsvarianten die Vor- und Nachteile angeben und anhand gegebener Anforderungen abwägen.
- Sie können verlinkte Listen einsetzen, um Datenstrukturen mit speziellen Zugriffsregeln wie *Stack* und *Queue* zu implementieren.

Iteratoren:

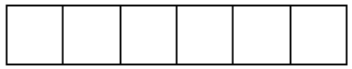
- Sie können zu einer Collection einen *Iterator* programmieren.
- Sie können die möglichen Konflikte erklären, die entstehen, wenn eine Collection während der Iteration verändert wird und können eine Lösung vorschlagen und programmieren.
- Sie können für eine Liste einen *ListIterator* implementieren.

Sequenzielle Anordnungen von Elementen im Java Collection Framework

- Gängigste Ablagen: Arrays, ArrayList, LinkedList

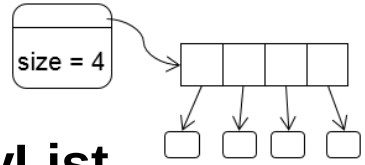


Vergleich: Arrays, ArrayList, LinkedList



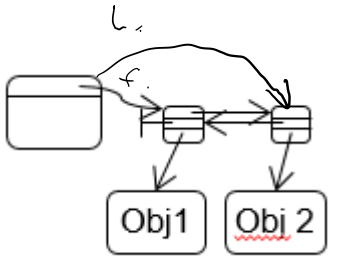
Arrays

- Fixe Grösse, reserviert
- Direkter Index-Zugriff
- Speichereffizient (falls (nahezu) voll)
- Erlaubt auch primitive Datentypen
- Einfügen zu Beginn teuer



ArrayList

- Nutzt Array, vergrössert sich automatisch
- Keine primitive Datentypen erlaubt
- Iterator add / remove in O(n)
- **Direkter Index-Zugriff**



LinkedList

- Grösse dynamisch angepasst
- Einfügen am Anfang in O(1)
- Braucht viel Speicherplatz
- Index-Zugriff in O(n)
- **Iterator kann Löschen / Einfügen in O(1)**

Vergleich: Arrays, ArrayList, LinkedList

Quelle: http://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist

	`get(int idx)`	`add(E e)`	`add(int idx, E e)`	`remove(int idx)`	`Iterator.remove()`	`ListIterator.add(E e)`
LinkedList	O(n)	O(1)	O(n)	O(n)	O(1)	O(1)
ArrayList	O(1)	O(1)*	O(n)	O(n)	O(n)	O(n)

* = Amortized

LinkedList: Grösserer Memory-Overhead (next und previous-Zeiger)

ArrayList: Default-Kapazität von 10