

Einführung in die Informatik

Reference Variables

Einfach und doppelt verkettete Listen

Wolfram Burgard

Einleitung

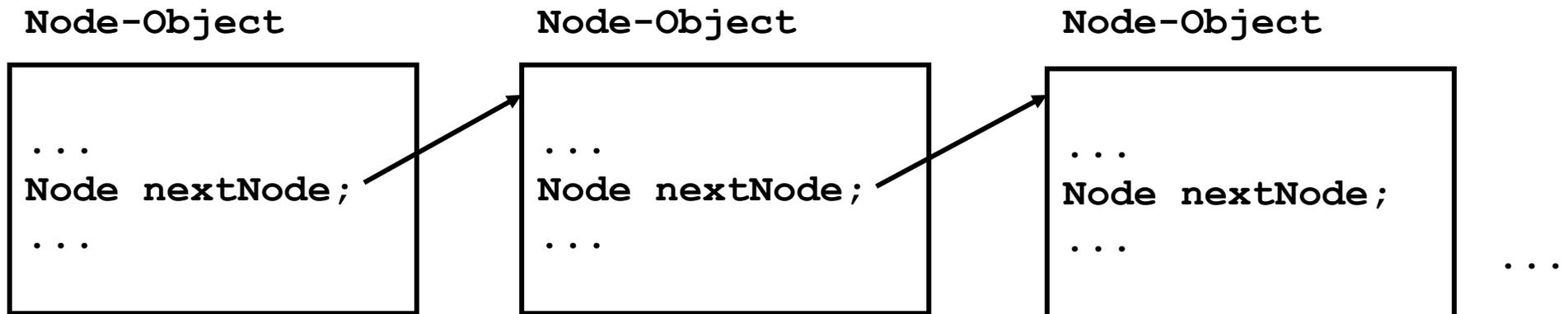
- Variablen enthalten Referenzen auf Objekte.
- Bei der Komposition von Objekten haben wir dies ausgenutzt und in **Instanzvariablen Referenzen auf Objekte** gespeichert.
- Dabei waren die Instanzvariablen immer Referenzen auf Objekte anderer Klassen.
- In diesem Kapitel werden wir den speziellen Fall betrachten, dass eine **Instanzvariable ein Objekt derselben Klasse referenziert**.
- Durch diesen Mechanismus lassen sich **Kollektionen** definieren, **die dynamisch** (d.h. zur Laufzeit) mit der Anzahl der zu repräsentierenden Objekte **wachsen können**.

Referenzen

- Eine **Referenz ist ein Verweis auf den Ort**, wo sich der Wert oder das Objekt befindet.
- **Auf der Maschinenebene ist die Referenz eine Speicheradresse**, an der der zugehörige Wert abgelegt ist.
- Variablen, deren Wert eine Referenz auf ein Objekt ist, heißen **Referenzvariablen**.
- Der spezielle Wert `null` symbolisiert dabei, dass die Variable auf **keine gültige Speicheradresse** verweist.
- Referenzvariablen werden auch als **Zeigervariablen, Zeiger oder Pointer** bezeichnet.

Verkettete Listen

- Bisher waren die referenzierten Objekte stets Instanzen anderer Klassen.
- Das besondere an verketteten Listen ist, dass Sie **Referenzen auf Objekte der eigenen Klasse** besitzen
- Dadurch können wir im Prinzip ganze **Ketten von Referenzen** definieren.

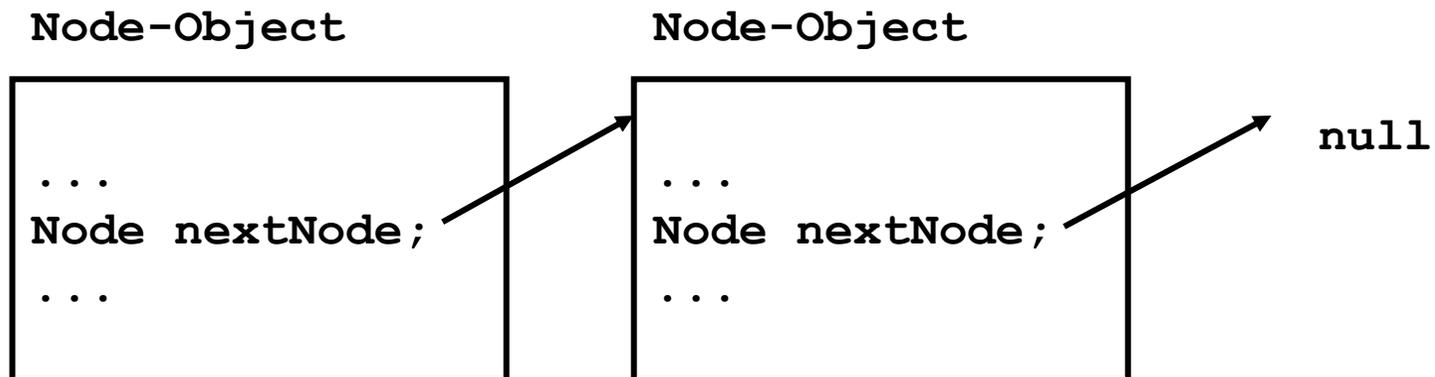


Verkettete Listen

- Beispiel einer Implementierung eines Listenelementes:

```
class Node {  
    ...  
    private Node nextNode;  
}
```

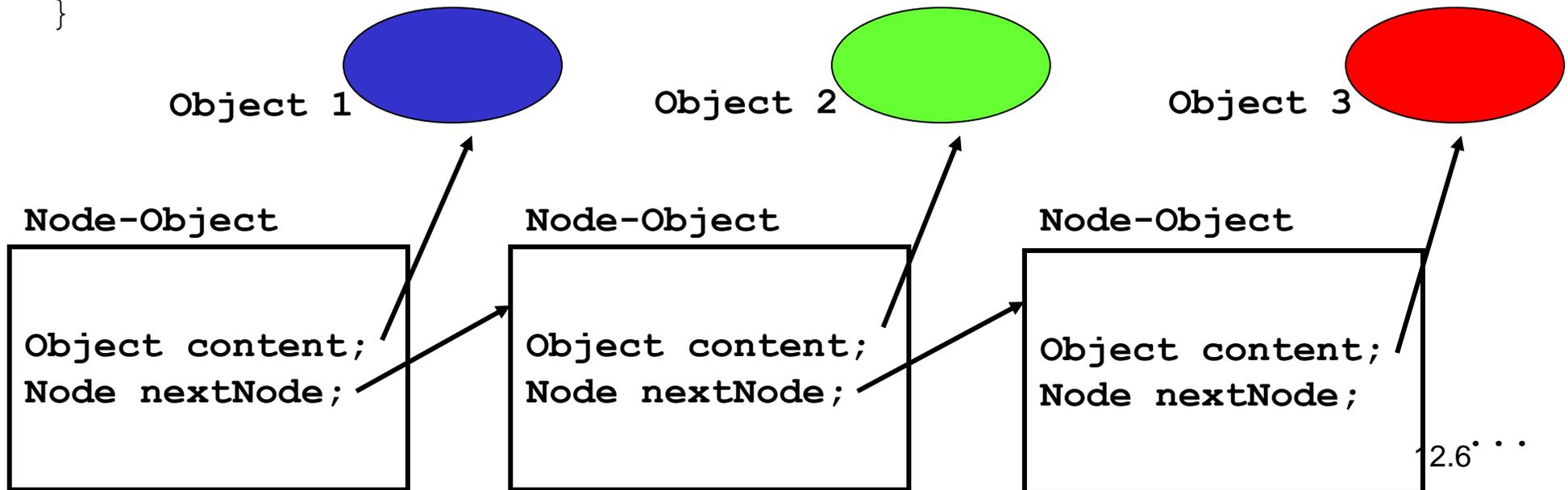
- **Verkettete Listen** lassen sich dadurch konstruieren, dass die Klasse eine **Instanzvariable** enthält, die auf **Objekte derselben Klasse referenziert**.
- Das Ende der Liste wird durch den Wert **null** markiert.



Inhalt von Listenelementen

- Um mit verketteten Listen Kollektionen zu realisieren, muss man in jedem Knoten einen Inhalt ablegen.
- Dies geschieht am allgemeinsten dadurch, dass man in jedem Knoten eine Referenz auf ein `Object`-Objekt in einer Instanzvariablen ablegt.

```
class Node {  
    ...  
    private Object content;  
    private Node nextNode;  
}
```



Listen und Knoten

- Es ist oft zweckmäßig, für Listen eine separate Klassen zu realisieren.
- In unserem Fall führen wir daher zusätzlich die Klasse **SingleLinkedList** für wie oben beschriebene, einfach verkettete Listen ein.
- Diese enthalten neben den üblichen Methoden für eine Liste auch die Referenz auf das erste Objekt, den so genannten **Kopf der Liste**.

```
public class SingleLinkedList {  
    ...  
    private Node head;  
}
```

Methoden für Knoten

Knoten sollen die folgenden Methoden liefern:

1. Inhalt setzen,
2. Inhalt lesen,
3. den nächsten Knoten erhalten,
4. die Referenz auf den nächsten Knoten setzen sowie
5. die Methode `toString`.

Die Prototypen der Methoden für die Klasse Node

```
public Node (Object o, Node n);  
public void setContent (Object o);  
public Object content ();  
public Node nextNode ();  
public void setNextNode (Node n);  
public String toString ();
```

Implementierung der Methoden für Node (1)

- Der Konstruktor erzeugt ein `Node`-Objekt und setzt eine Referenz auf ein `Object`-Objekt.
- Gleichzeitig wird die Referenz auf das Nachfolgeelement gesetzt.

```
public Node (Object o, Node n) {  
    this.content = o;  
    this.nextNode = n;  
}
```

Implementierung der Methoden für Node (2)

- Die Methode `content` liefert das im Knoten abgelegt Objekt.

```
public Object content() {  
    return this.content;  
}
```

- Mithilfe der Methode `setContent` kann der Inhalt eines Knotens gesetzt werden:

```
public void setContent(Object o) {  
    this.content = o;  
}
```

Implementierung der Methoden für Node (3)

- Die Methoden `nextNode` liefert als Ergebnis den Inhalt der Instanzvariablen `nextNode`, d.h. die im Knoten gespeicherte Referenz auf das Nachfolgeelement.

```
public Node nextNode() {  
    return this.nextNode;  
}
```

- Mit `setNextNode` kann diese Referenz gesetzt werden:

```
public void setNextNode(Node n) {  
    this.nextNode = n;  
}
```

- Die Methode `toString`:

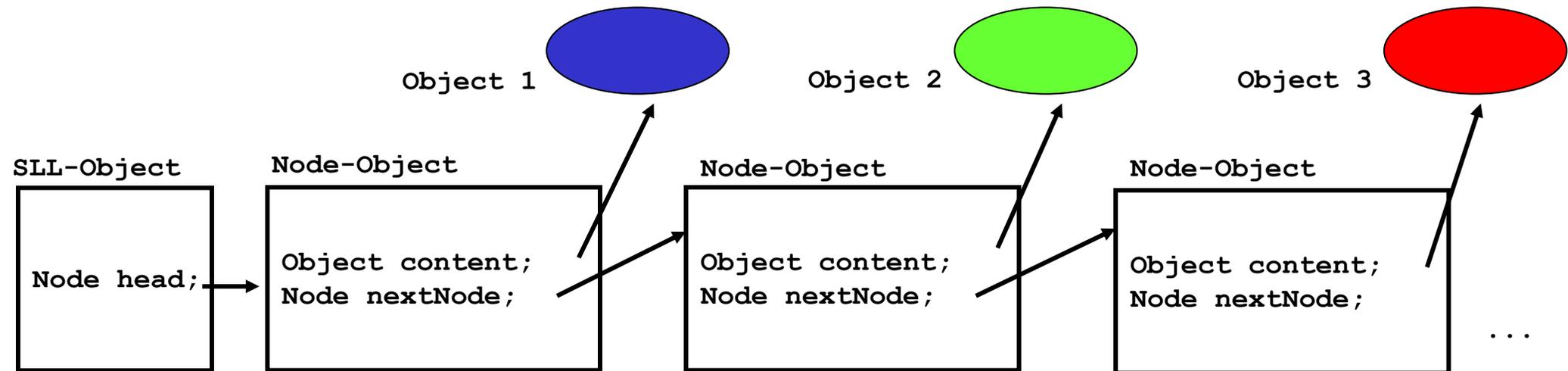
```
public String toString() {  
    return this.content.toString();  
}
```

Die komplette Klasse Node

```
class Node {
    public Node (Object o, Node n) {
        this.content = o;
        this.nextNode = n;
    }
    public Object content() {
        return this.content;
    }
    public void setContent(Object o) {
        this.content = o;
    }
    public Node nextNode() {
        return this.nextNode;
    }
    public void setNextNode(Node n) {
        this.nextNode = n;
    }
    public String toString() {
        return this.content.toString();
    }
    Object content;
    Node nextNode;
}
```

Liste

- Objekte der Klasse Node zum Speichern einzelner Elemente.
- Die Node Objekte enthalten jeweils eine Referenz auf das nächste Element. Diese ist auch ein Objekt der Klasse Node.
- Listen-Klasse (SingleLinkedList, SLL) zum Verwalten der Liste.



Methoden für Listen

Ähnlich wie für andere Kollektionen auch sollen Listen die folgenden Methoden zur Verfügung stellen:

1. Test, ob die Liste leer ist,
2. Einfügen eines neuen Listenelementes am Anfang,
3. Einfügen eines neuen Listenelementes am Ende,
4. Einfügen eines neuen Listenelementes an einer beliebigen Stelle in der Liste,
5. Löschen eines Listenelementes,
6. Suchen eines Knotens, der ein gegebenes Objekt enthält,
7. Invertieren der Reihenfolge der Listenelemente und
8. Aufzählen aller in einer Liste enthaltenen Objekte.

Die Prototypen der Methoden der Klasse SingleLinkedList

```
public SingleLinkedList();  
public boolean isEmpty();  
public void insertHead(Object o);  
public void insertTail(Object o);  
public void insertAfterNode(Object o, Node node);  
private void removeNextNode(Node node);  
public void removeFirstNode();  
public Node searchNode(Object o)  
public void reverseList();  
public String toString();
```

Der Konstruktor und die Methode `isEmpty`

Der **Konstruktor erzeugt eine leere Liste**:

```
public SingleLinkedList() {  
    this.head = null;  
}
```

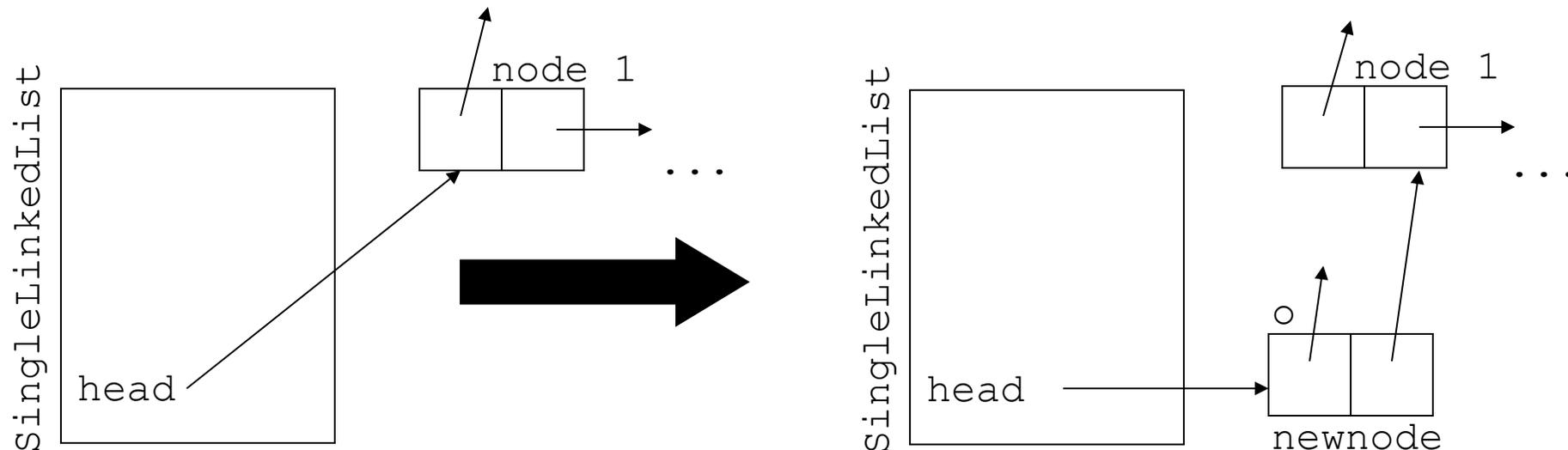
Die Methode **`isEmpty`** liefert genau dann **`true`**, wenn der Kopf der Liste den Wert **`null`** hat:

```
public boolean isEmpty() {  
    return (this.head == null);  
}
```

Einfügen eines neuen Elementes am Anfang der Liste

Hierbei müssen wir

1. einen neuen Knoten erzeugen,
2. die Referenzvariable für den Listenkopf auf diesen Knoten setzen und
3. in diesem Knoten `nextNode` auf das zuvor erste Element setzen.

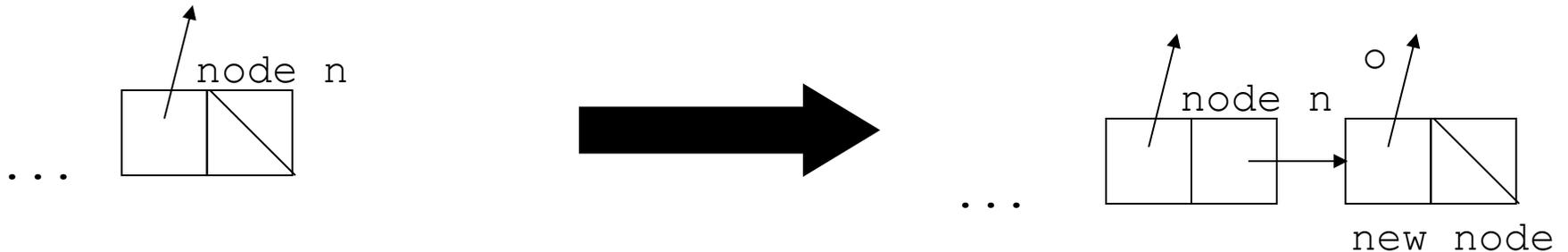


```
public void insertFirst(Object o) {  
    this.head = new Node(o, this.head);  
}
```

Einfügen eines neuen Elementes am Ende der Liste

Um ein neues Element am Ende einzufügen, müssen wir

1. einen neuen Knoten erzeugen,
2. zum Ende der Liste laufen und das neue Element anfügen und
3. den Spezialfall beachten, dass die Liste leer sein könnte.

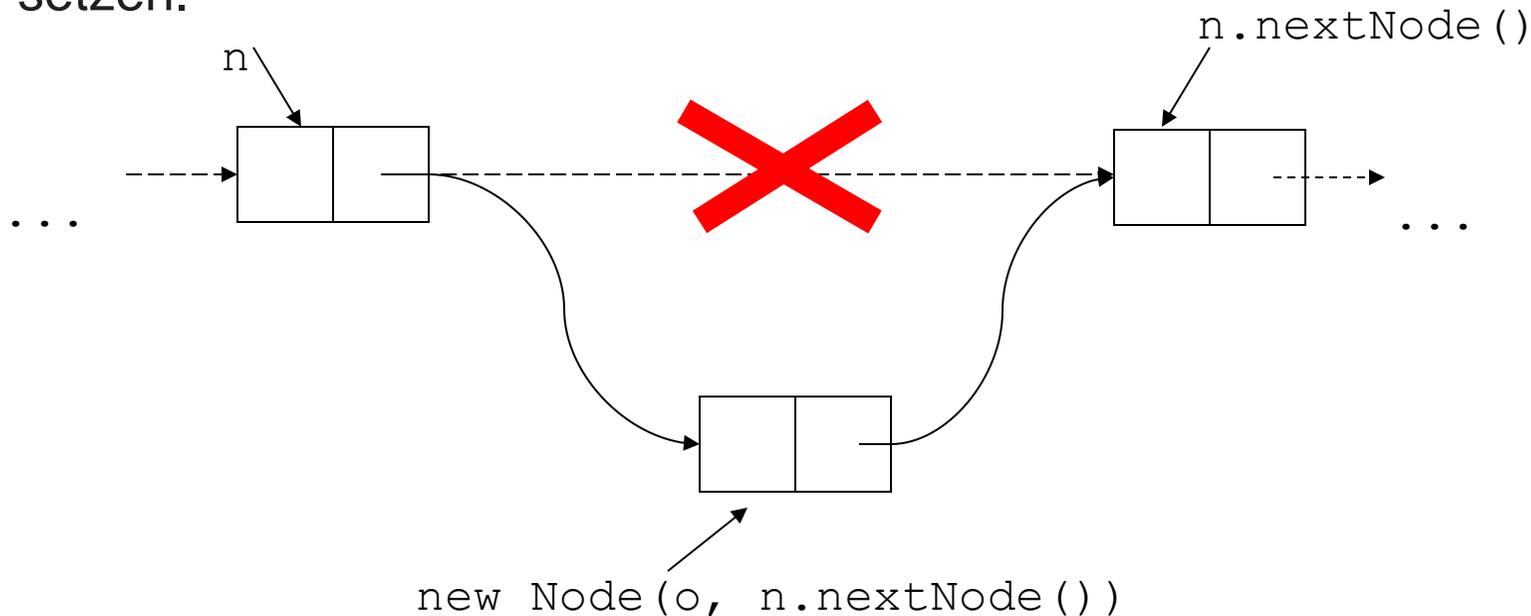


Die Methode insertLast

```
public void insertLast(Object o) {
    if (this.isEmpty()) {           /* list is empty */
        this.insertFirst(o);
    }
    else {
        Node tmp = this.head;
        while (tmp.nextNode() != null)
            tmp = tmp.nextNode();
        tmp.setNextNode(new Node(o, null));
    }
}
```

Einfügen eines neuen Knotens nach einem Knoten

1. Nachfolgereferenz des neuen Knotens auf den Nachfolgeknoten des aktuellen Knotens setzen
2. Nachfolgereferenz des aktuellen Knotens auf den neuen Knoten setzen.



```
public void insertAfterNode(Object o, Node n) {  
    n.setNextNode(new Node(o, n.nextNode()));  
}
```

Löschen eines Listenelementes

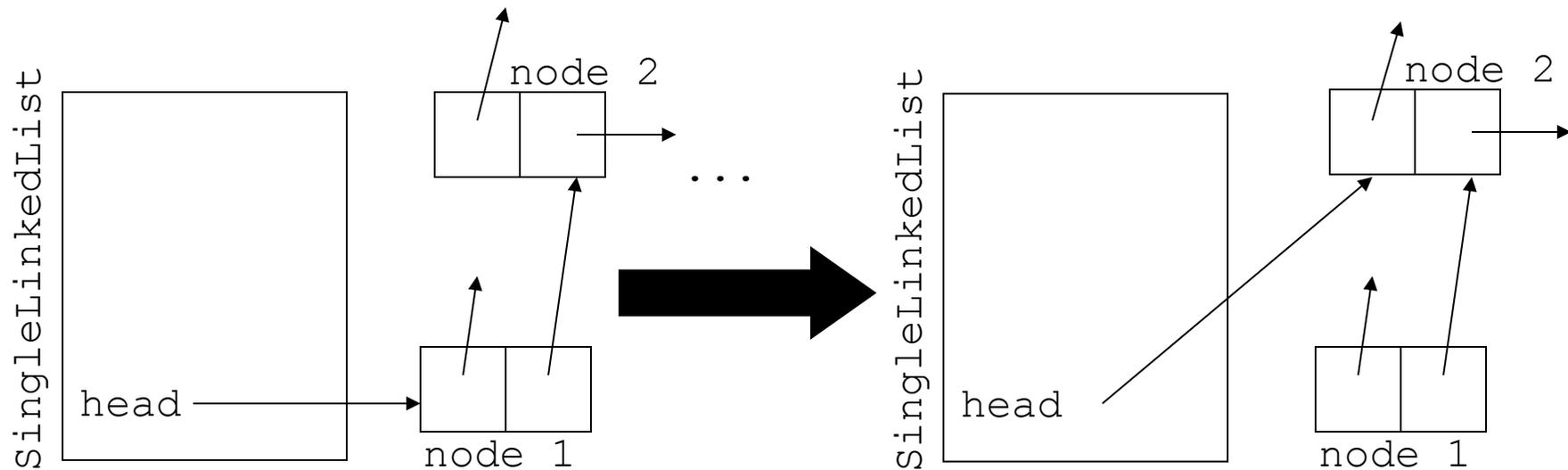
Hierbei unterscheiden wir zwei Fälle:

1. Löschen des ersten Elementes.
2. Löschen des Nachfolgeelementes eines gegebenen Elementes.

Dabei müssen wir uns nicht um die nicht mehr referenzierten `Node`-Objekte kümmern. Die Freigabe des Speichers übernimmt das Java-System mit seiner **automatischen Garbage Collection**.

Löschen des ersten Elementes einer Liste

- Wenn wir das erste Element einer Liste löschen, genügt es, den Wert der Instanzvariablen `head` auf das zweite Listenelement zu setzen.
- Allerdings darf der Wert von `head` nicht `null` sein.



```
void removeFirstNode() {  
    if (!this.isEmpty())  
        this.head = this.head.nextNode();  
}
```

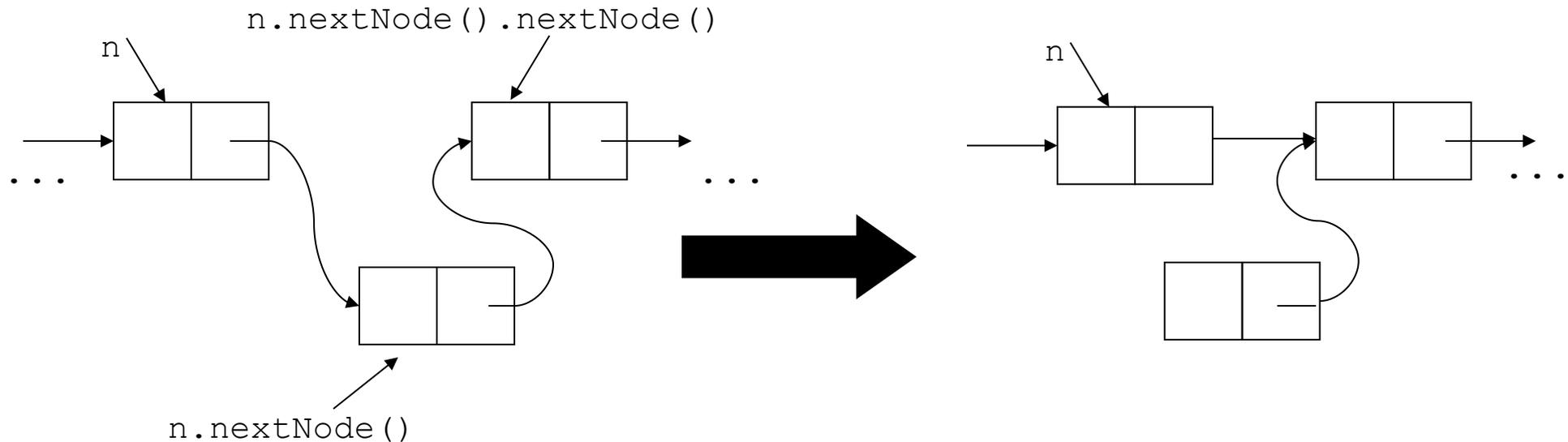
Der andere Fall: `removeNextNode`

- Wir können ein Listenelement nur dann Löschen, wenn wir auch den Vorgänger kennen, da wir dort die Referenzvariable `nextNode` auf den Nachfolger des Nachfolgeknotens setzen müssen:

```
node.setNextNode (node.nextNode () .nextNode () ) ;
```

- Allerdings geht dies nur, wenn das Listenelement, nach dem wir löschen wollen, nicht das letzte Element der Liste ist.

Die Methode `removeNextNode`



```
private void removeNextNode(Node n) {  
    if (n.nextNode() != null)  
        n.setNextNode(n.nextNode().nextNode());  
}
```

Suchen eines Listenelementes mit einem bestimmten Inhalt

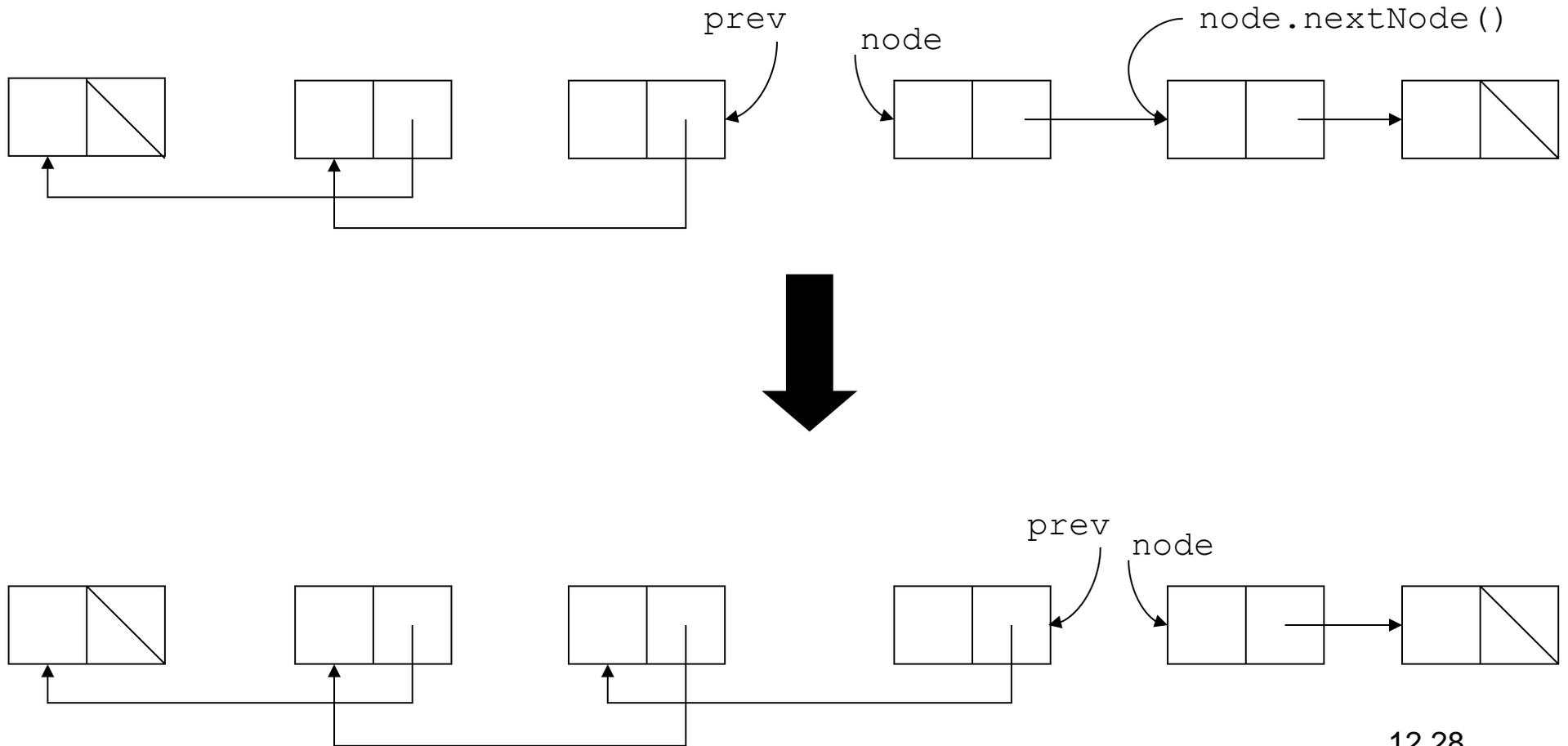
- Durchlaufen der Liste.
- Rückgabe der Referenz auf den Knoten, der das gesuchte Objekt enthält.

```
public Node searchNode(Object o) {  
    Node n = this.head;  
    while (n != null && !n.content().equals(o))  
        n = n.nextNode();  
    return n;  
}
```

Invertieren einer Liste

- Da wir **einfach verkettete Listen** immer nur in einer Richtung durchlaufen können, wäre eine **Invertierung durch Vertauschen der Elemente** (wie wir das bei Vektoren realisiert haben) **zu aufwendig**.
- Daher können wir Listen effizient nur dadurch **invertieren**, dass wir die **Referenzen in den Listenelementen geeignet umsetzen**.
- Im Prinzip muss die **Nachfolgereferenz** in einem Knoten nur so gesetzt werden, dass sie das **Vorgängerelement referenziert**.
- Um dies zu realisieren benötigen wir daher **zwei** Referenzvariablen: Eine, die für den **Anfang des noch zu invertierenden Restes** der Liste und eine für das ehemalige Vorgängerelement, d.h. den **Anfang des bereits invertierten Teils** der Liste.
- **Beim letzten Listenelement** sind wir **fertig**.
- In diesem Fall muss die **Referenzvariable head** auf den **zuletzt besuchten Knoten** gesetzt werden.

Das Verfahren zum Invertieren einer Liste



Eine rekursive Implementierung

```
private void reverseRecursive(Node node, Node prev) {
    Node next = node.nextNode();
    if (next == null)
        this.head = node;
    else {
        reverseRecursive(next, node);
    }
    node.setNextNode(prev);
}

public void reverseList() {
    if (!this.isEmpty())
        reverseRecursive(this.head, null);
}
```

Die Methode toString

- Um die Methode `toString` zu realisieren, müssen wir einmal die Liste durchlaufen.
- Dabei müssen wir ein entsprechendes `String`-Objekt zusammensetzen.

```
public String toString() {
    String str = "[";
    Node tmp = this.head;
    while (tmp != null) {
        str += tmp.content().toString();
        tmp = tmp.nextNode();
        if (tmp != null)
            str += ", ";
    }
    return str+"]";
}
```

Ein Iterator-Interface für Listen

- Um Programmieren eine **komfortable Möglichkeit** zur Verfügung zu stellen, **Listendurchläufe** zu realisieren, stellen wir ein **Iterator-Interface** zur Verfügung.
- Da wir ggf. mehrere Iteratoren unabhängig voneinander nutzen wollen, realisieren wir eine **Hilfsklasse**

```
class SingleLinkedListIterator implements Iterator {  
    ...  
}
```

- Für **jeden Iterator** verwenden wir ein **eigenes Objekt dieser Klasse**.
- Darin speichern wir den **Zustand des Iterator-Objektes zwischen einzelnen next-Aufrufen**.

Das Interface Iterator

```
interface Iterator {  
    Object next();  
    boolean hasNext();  
    void remove();  
}
```

Die Klasse `SingleLinkedListIterator`

- Zur Speicherung des Zustands eines `Iterators` für Listen ist eine Referenz auf den nächsten zu besuchenden Knoten hinreichend.
- Sie erlaubt den Zugriff auf das Objekt im nächsten Element.
- Falls die Referenz auf das nächste Element den Wert `null` hat, gibt es keine weiteren Elemente mehr, da wir am Ende der Liste angekommen sind.

Implementierung der Klasse SingleLinkedListIterator

```
class SingleLinkedListIterator implements Iterator {
    public SingleLinkedListIterator(Node node) {
        this.node = node;
    }
    public boolean hasNext() {
        return (this.node != null);
    }
    public Object next() {
        Object o = node.getContent();
        this.node = this.node.getNextNode();
        return o;
    }
    public remove() { // to be implemented }

    private Node node;
}
```

Verwendung der Klasse `SingleLinkedListIterator`

- Um auf die übliche Weise ein `Iterator`-Objekt zu erzeugen, muss unsere `List`-Klasse ein Methode `iterator()` bereitstellen.
- Eigentlich können wir nur ein `SingleListIterator`-Objekt zurückgeben.
- Der Programmierer kennt jedoch nur die Klasse `Iterator`, d.h. Er wird folgendes hinschreiben wollen:

```
Iterator e = list.iterator();
while (e.hasNext()) {
    Object o = (Object) e.next();
    ...
}
```

Lösung: Erweiterung Is-a-Beziehung auf Interfaces

- Implementiert eine Klasse ein Interface so besteht **zwischen den Objekten und dem Interface eine Is-a-Beziehung**, d.h. die Klasse wird behandelt wie eine Sub-Klasse.
- Damit ist folgende Definition innerhalb der Klasse `SingleLinkedList` zulässig:

```
public Iterator iterator() {  
    return new SingleLinkedListIterator(this.head);  
}
```

Suche eines Objektes in einer Liste

- Das Suchen eines Objektes in einer Liste kann man über einen Iterator realisieren.
- **Annahme:** die Klasse `SingleLinkedList` stellt einen entsprechenden `Iterator` zur Verfügung

```
static boolean contains(Object o, SingleLinkedList l) {  
    Iterator e = l.iterator();  
    while (e.hasNext()) {  
        Object content = e.next();  
        if (content.equals(o))  
            return true;  
    }  
    return false;  
}
```

Auswirkung des dynamischen Bindens auf die Suche

- Innerhalb der Suchmethode verwenden wir die Methode `equals`.
- Wegen des **dynamische Bindens** wird **erst zur Laufzeit festgelegt, welche Methode tatsächlich ausgeführt wird.**

```
Object i1 = new Integer(1), i2 = new Integer(1);  
list.insertLast(i1);  
Object o1 = new Object(), o2 = new Object();  
list.insertLast(o1);  
System.out.println(contains(i2, list));  
System.out.println(contains(o2, list));
```

- Während die `equals`-Methode der Klasse `Integer` den Inhalt vergleicht, überprüft `Object.equals` nur die Referenzen. Daher ist die Suche im ersten Fall erfolgreich. Im zweiten Fall scheitert sie.

Anwendung der Klasse SingleLinkedList

```
import java.util.Iterator;
public class SingleLinkedListTest {
    public static void main(String args[]) {
        /* variables */
        SingleLinkedList list;
        /* create list */
        list = new SingleLinkedList();

        /* insert elements */
        list.insertFirst(new Integer(2));
        Node n = list.searchNode(new Integer(2));
        list.insertAfterNode(new Integer(1), n);
        list.insertLast(new Integer(3));
        list.insertFirst(new Integer(4));
        list.insertLast(new Integer(5));
        System.out.println(list);
        list.removeFirstNode();
        System.out.println(list);

        /* search for elements */
        System.out.println(new Integer(1)
            + " in the list: "
            + (list.searchNode(new Integer(1)) != null));
        System.out.println(new Integer(3)
            + " in the list: "
            + (list.searchNode(new Integer(3)) != null));

        /* remove elements */
        list.remove(new Integer(1));
        list.remove(new Integer(5));
        System.out.println(list);
        /* searchNode for elements */
        System.out.println(new Integer(1)
            + " in the list: "
            + (list.searchNode(new Integer(1)) != null));
    }
}
```

```
System.out.println(new Integer(3)
    + " in the list: "
    + (list.searchNode(new Integer(3))
        != null));
/* remove elements */
list.remove(new Integer(1));
System.out.println(new Integer(3)
    + " in the list: "
    + (list.searchNode(new Integer(3))
        != null));
/* inserting elements */
list.insertFirst(new Integer(2));
System.out.println(list);
/* reverse list */
list.reverseList();
list.reverseList();
list.reverseList();
System.out.println(list);
/* test the iterator */
System.out.println("testing iterator:");
Iterator i = list.iterator();
while (i.hasNext()) {
    Object o = (Object) i.next();
    System.out.println(o.toString());
}
Object i1 = new Integer(1),
        i2 = new Integer(1);
list.insertLast(i1);
Object o1 = new Object(),
        o2 = new Object();
list.insertLast(o1);
System.out.println(list.searchNode(i2));
System.out.println(list.searchNode(o2));
System.out.println("DONE");
}
```

Ausgabe des Programms

```
insertingFirst 2      insertingFirst 2
insertingAfter 1     [2, 2, 3]
insertingLast 3      reversing list
insertingFirst 4     reversing list
insertingLast 5      reversing list
[4, 2, 1, 3, 5]      [3, 2, 2]
removing first node  testing iterator:
[2, 1, 3, 5]         3
1 in the list: true   2
3 in the list: true   2
removing 1           insertingLast 1
removing 5           insertingLast java.lang.Object@80ab1d8
[2, 3]              1
1 in the list: false null
3 in the list: true  DONE
```

Ein JUnit Test für SingleLinkedList

```
import org.junit.Test;
import org.junit.Assert;

/**
 * My Standard Test class.
 * @author Cyrill Stachniss
 */
public class SingleLinkedListTest {
    /**
     * My testcase.
     */
    @Test public void testSLLSimple() {
        SingleLinkedList list = new SingleLinkedList();
        Assert.assertEquals( true, list.isEmpty() );

        Integer i = new Integer(2);
        list.insertFirst(i);
        Assert.assertEquals( i, list.searchNode(i).content() );
        Assert.assertEquals( false, list.isEmpty() );
    }
}
```

Ein JUnit Test für SingleLinkedList

```
Terminal — bash — 85x21

stachnis@localhost:~/tmp/MyProject> ant test
Buildfile: /Users/stachnis/tmp/MyProject/build.xml

compile:

test:
[junit] Testsuite: MyCounterTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.013 sec
[junit]
[junit] Testcase: testMyCounter took 0.001 sec
[junit] Testcase: testInc took 0 sec
[junit] Testcase: testReset took 0 sec
[junit] Testsuite: SingleLinkedListTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.012 sec
[junit]
[junit] Testcase: testSLLSimple took 0.002 sec

BUILD SUCCESSFUL
Total time: 1 second
stachnis@localhost:~/tmp/MyProject> 
```

Aufwand einiger Listenoperationen

Operation	SingleLinkedList	ArrayList
Einfügen am Anfang	$O(1)$	$O(n)$
Einfügen am Ende	$O(n)$	$O(1)$ (a)
Einfügen an gegebener Stelle	$O(1)$	$O(n)$
Suchen	$O(n)$	$O(n)$
Suchen in sortierter Kollektion	$O(n)$	$O(\log n)$
Invertieren	$O(n)$	$O(n)$

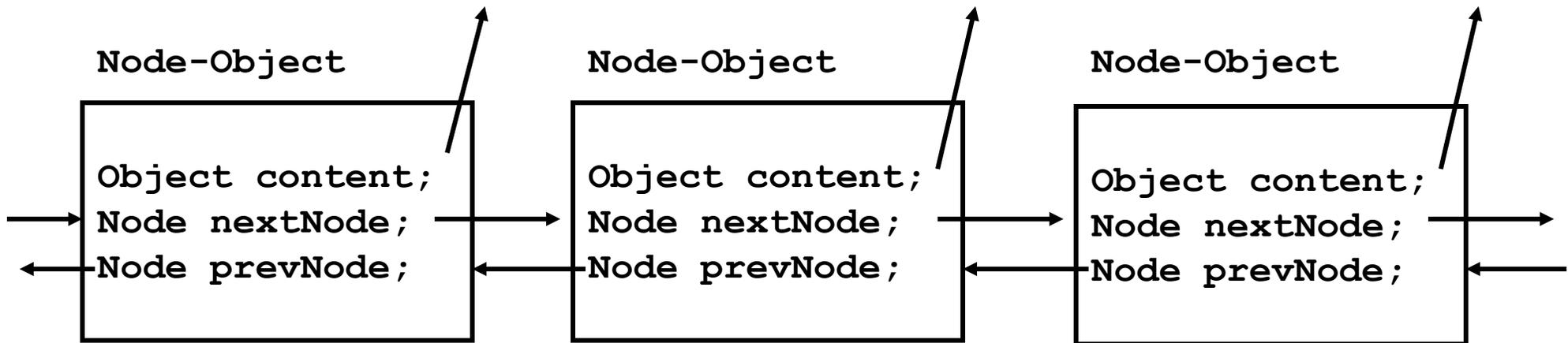
Hinweis zu (a): Nur falls die ArrayList noch Platz für neue Elemente hat, sonst $O(n)$.

Doppelt verkettete Listen

- **Einfach verkettete Listen** haben den Nachteil, dass man sie **nur in einer Richtung durchlaufen** kann.
- Darüber hinaus kann man ein **referenziertes Listenelement nicht unmittelbar löschen**, da man von diesem Element **keinen Zugriff auf das Vorgängerelement** hat.
- **Doppelt verkettete Listen** umgehen dieses Problem, indem sie in jedem Knoten **zusätzlich noch eine Referenz auf den Vorgängerknoten** speichern.

Die Klasse Node für doppelt verkettete Listen

- Im Gegensatz zu einfach verketteten Listen haben **doppelt verkettete Listen** in den Knoten eine **zusätzliche Instanzvariable für die Referenz auf den Vorgängerknoten**



```
class Node {  
    ...  
    private Object content;  
    private Node nextNode;  
    private Node prevNode;  
}
```

Methoden für die Klasse Node

- Die Methoden für Knoten in doppelt verketteten Listen sind eine einfache Erweiterung der entsprechenden für einfach verkettete.
- Allerdings kommen noch einige Methoden für das Vorgängerelement hinzu.
- Der Konstruktor beispielsweise muss nun folgendermaßen realisiert werden:

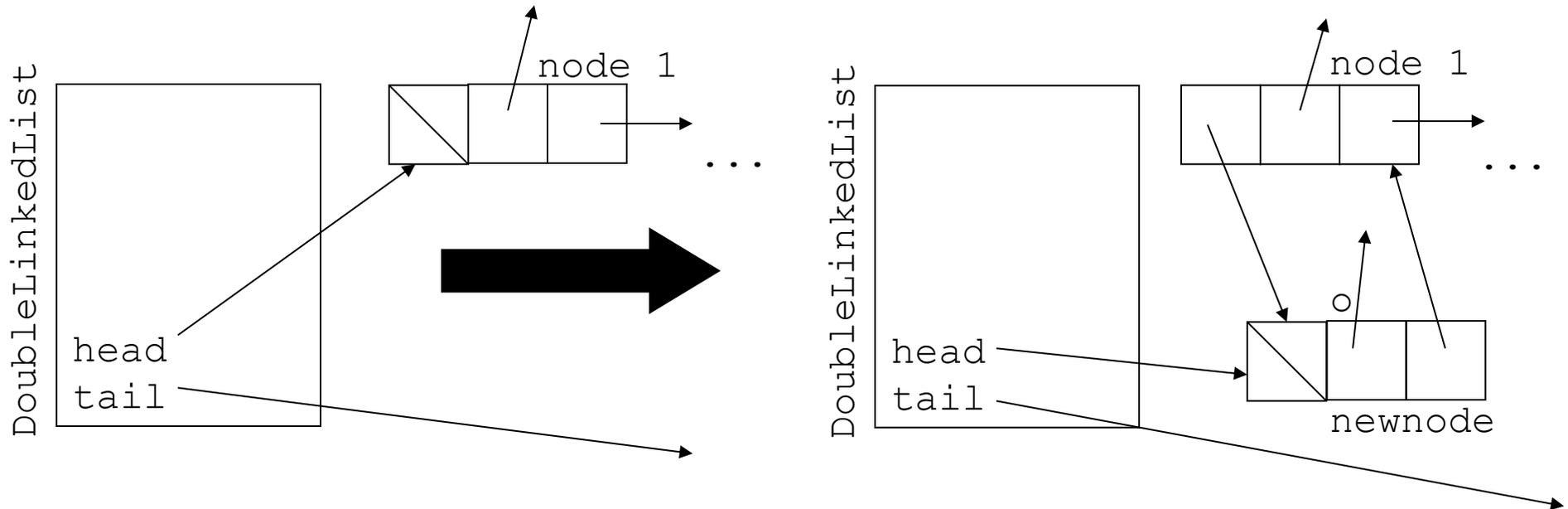
```
class Node {
    public Node (Object o, Node prev, Node next) {
        this.content = o;
        this.nextNode = next;
        this.prevNode = prev;
    }
    ... // Rest analog
}
```

Die Klasse `DoubleLinkedList`

- Die Klasse `DoubleLinkedList` hat im wesentlichen dieselben Methoden wie die Klasse `SingleLinkedList`.
- Bei der Realisierung der Methoden muss man allerdings darauf achten, dass stets auch die Referenz auf den Vorgängerknoten korrekt gesetzt wird.
- Außerdem wollen wir in der Klasse `DoubleLinkedList` auch eine **Instanzvariable `tail`** für das **letzte Listenelement** ablegen.

```
public class DoubleLinkedList {  
    ...  
    protected Node head;  
    protected Node tail;  
}
```

Beispiel: Die Methode insertHead



```
public void insertHead(Object o) {  
    if (this.isEmpty())  
        this.head = this.tail = new Node(o, null, null);  
    else {  
        this.head.setPreviousNode(new Node(o, null, this.head));  
        this.head = this.head.previousNode();  
    }  
}
```

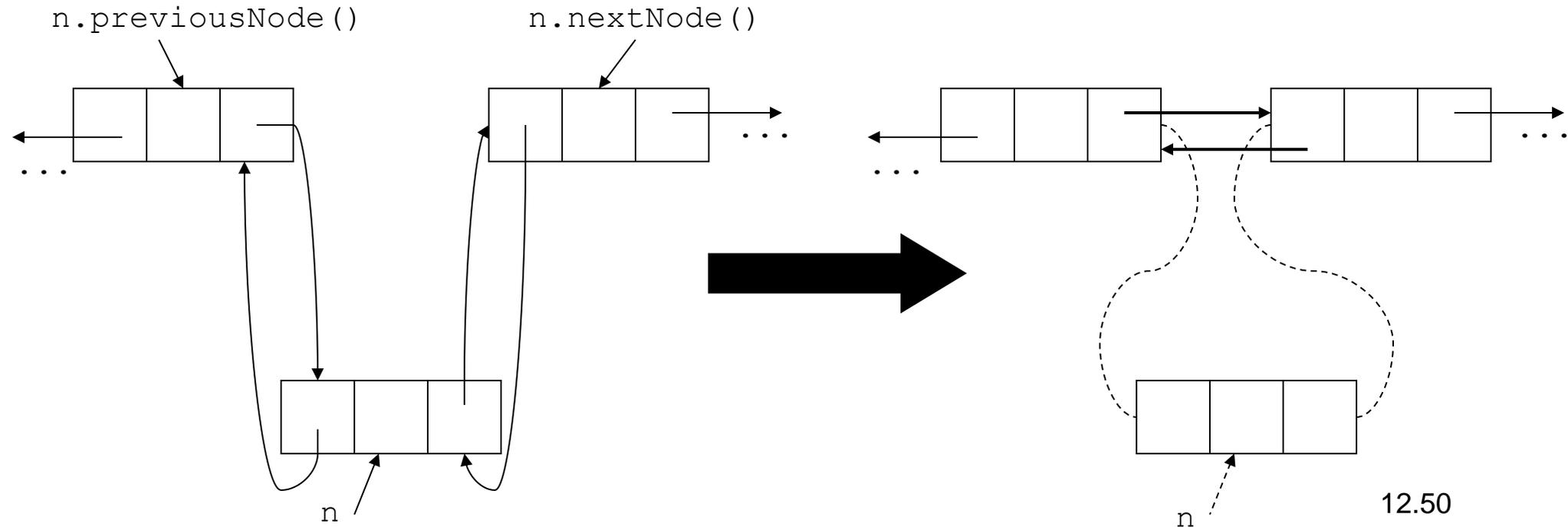
Die Methode `insertTail`

- Dadurch, dass wir jetzt eine Referenz auf das letzte Element haben, können wir wesentlich effizienter am Ende einfügen:
- Diese Operation ist symmetrisch zum Einfügen am Anfang.

```
public void insertTail(Object o) {
    if (this.isEmpty())
        this.insertHead(o);
    else {
        this.tail.setNextNode(new Node(o, this.tail, null));
        this.tail = this.tail.nextNode();
    }
}
```

Die Methode `removeNode`

- Da wir bei doppelt verketteten Listen das Vorgängerelement und das Nachfolgerelement erreichen können, haben wir die Möglichkeit, referenzierte Listenelemente direkt zu löschen.
- Dabei müssen wir jedoch die Fälle berücksichtigen, dass sich das zu löschende Element am Anfang oder am Ende der Liste befinden kann.



Die Implementierung der Methode `removeNode`

```
public void removeNode(Node n) {
    Node nextNode = n.nextNode();
    Node prevNode = n.previousNode();
    if (this.head == n)
        this.head = nextNode;
    if (this.tail == n)
        this.tail = prevNode;
    if (prevNode != null)
        prevNode.setNextNode(nextNode);
    if (nextNode != null)
        nextNode.setPreviousNode(prevNode);
}
```

Die Methoden `removeHead` und `removeTail`

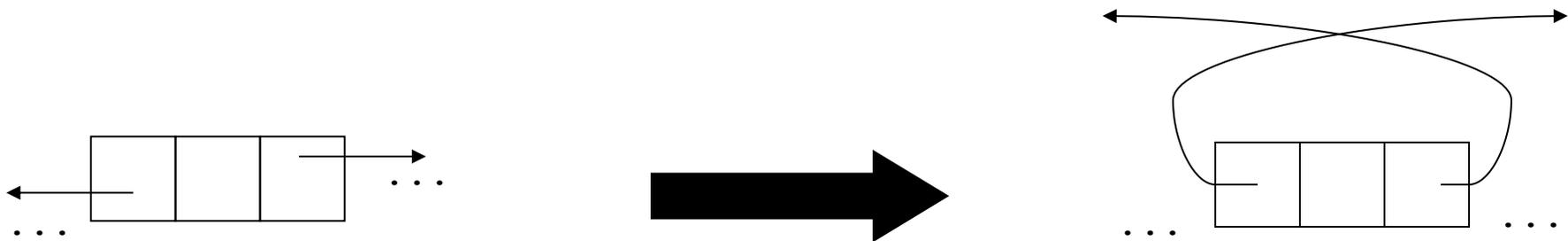
Diese Methoden lassen sich nun sehr leicht realisieren:

```
public void removeHead() {  
    this.removeNode(this.head);  
}
```

```
public void removeTail() {  
    this.removeNode(this.tail);  
}
```

Invertieren einer doppelt verketteten Liste

- Das Invertieren einer doppelt verketteten Liste ist ebenfalls deutlich einfacher als bei einer einfach verketteten Liste.
- Wegen der Symmetrie brauchen genügt es, in jedem Element die Vorgänger- und Nachfolgerreferenzen zu vertauschen.
- Zusätzlich müssen die Werte von `head` und `tail` getauscht werden.



Implementierung von reverse

```
public void reverse() {
    Node tmp = this.head;
    while (tmp != null) {
        // swap prev and next
        Node next = tmp.nextNode();
        tmp.setNextNode(tmp.previousNode());
        tmp.setPreviousNode(next);
        tmp = next;
    }
    // swap head and tail
    tmp = this.head;
    this.head = this.tail;
    this.tail = tmp;
}
```

Aufwand einiger Listenoperationen im Vergleich

SingleLinkedList: SLL
DoubleLinkedList: DLL
ArrayList: AL

Operation	SLL	DLL	AL
Einfügen am Anfang	$O(1)$	$O(1)$	$O(n)$
Einfügen am Ende	$O(n)$	$O(1)$	$O(1)$
Einfügen an gegebener Stelle	$O(1)$	$O(1)$	$O(n)$
Suchen	$O(n)$	$O(n)$	$O(n)$
Suchen in sortierter Kollektion	$O(n)$	$O(n)$	$O(\log n)$
Invertieren	$O(n)$	$O(n)$	$O(n)$

Beispiel Programme

- In den Beispielprogrammen zur Vorlesung finden sich die Programme
 - SingleLinkedList.java
 - SingleLinkedListTest.java

 - DoubleLinkedListSimple.java
 - DoubleLinkedListSimpleTest.java

 - DoubleLinkedList.java
 - DoubleLinkedListTest.java