



# Dynamische Datentypen

---

Listen, Dynamischer Speicher,  
Funktionalität, Copy-Konstruktor,  
Zuweisungsoperator, Destruktor,  
Konzept Dynamischer Datentyp



# Container für eine Folge gleichartiger Daten

---

- Bisher: Felder (`T[]`, `std::array<T>`,  
`std::vector<T>`)



# Container für eine Folge gleichartiger Daten

---

- Bisher: Felder  
(`T[]`, `std::vector<T>`)
- Zusammenhängender Speicherbereich,  
wahlfreier Zugriff (auf i-tes Element)

<b>1</b>	<b>5</b>	<b>6</b>	<b>2</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	----------	----------	----------	----------



# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>	<b>2</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	----------	----------	----------	----------

↑  
**8**

Wollen wir hier einfügen,  
müssen wir alles rechts  
davon explizit verschieben



# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>	<b>2</b>		<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	--	----------	----------	----------	----------

↑  
**8**

Wollen wir hier einfügen,  
müssen wir alles rechts  
davon explizit verschieben



# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>	<b>2</b>	<b>8</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------

Wollen wir hier einfügen,  
müssen wir alles rechts  
davon explizit verschieben

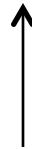


# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>	<b>2</b>	<b>8</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------



Wollen wir hier löschen,  
müssen wir alles rechts  
davon explizit verschieben



# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>		<b>8</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	--	----------	----------	----------	----------	----------

Wollen wir hier löschen,  
müssen wir alles rechts  
davon explizit verschieben





# Probleme mit Feldern

---

- Man kann keine Elemente “in der Mitte” einfügen oder “in der Mitte” löschen

<b>1</b>	<b>5</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>7</b>	<b>7</b>	<b>0</b>
----------	----------	----------	----------	----------	----------	----------	----------

Wollen wir hier löschen,  
müssen wir alles rechts  
davon explizit verschieben



# Lösung: Listen

---

- Container für eine Folge von Daten gleichen Typs



# Lösung: Listen

---

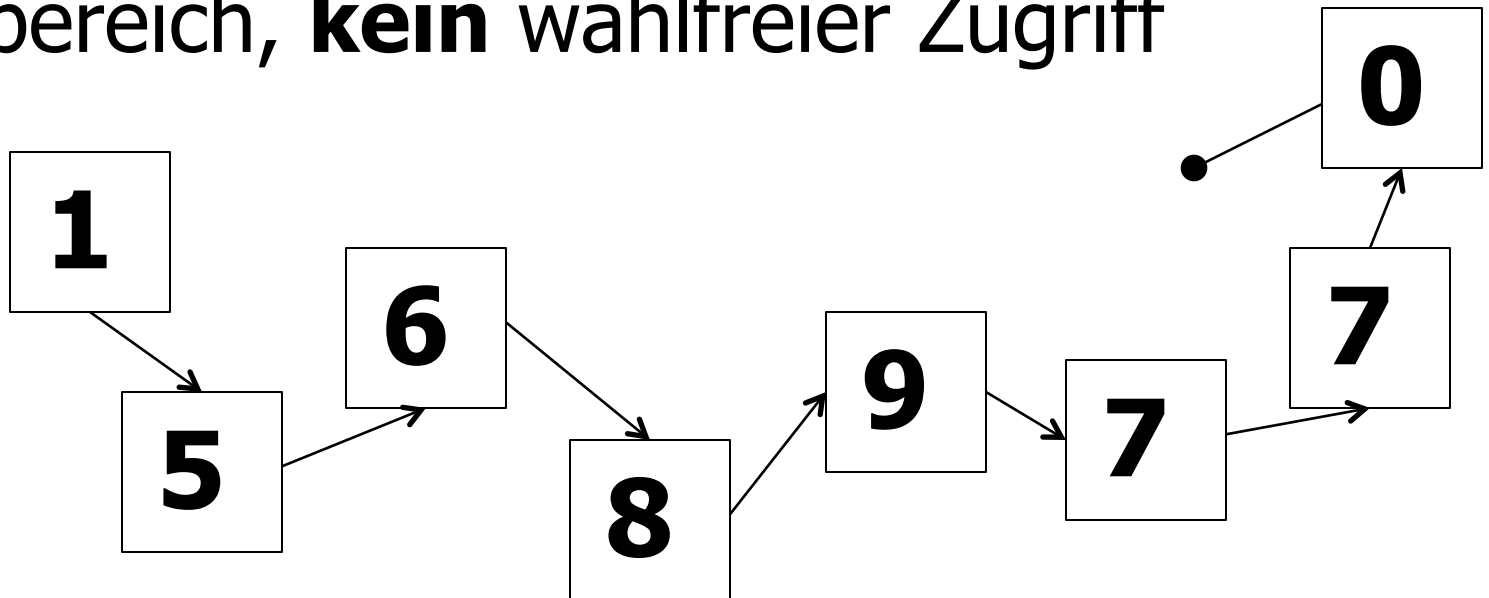
- Container für eine Folge von Daten gleichen Typs
- **Kein** zusammenhängender Speicherbereich, **kein** wahlfreier Zugriff



# Lösung: Listen

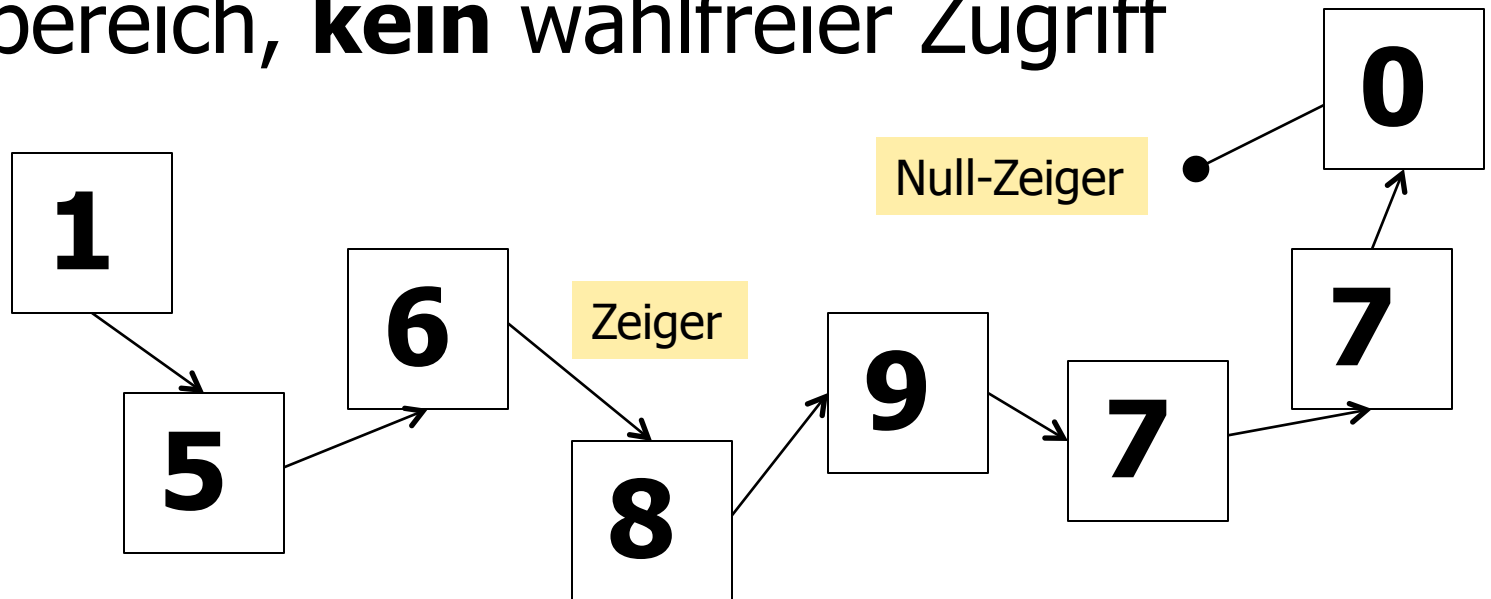
---

- Container für eine Folge von Daten gleichen Typs
- **Kein** zusammenhängender Speicherbereich, **kein** wahlfreier Zugriff



# Lösung: Listen

- Container für eine Folge von Daten gleichen Typs
- **Kein** zusammenhängender Speicherbereich, **kein** wahlfreier Zugriff





# Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

---

```
class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();

    ...

private:
    Node* head_;

    ...

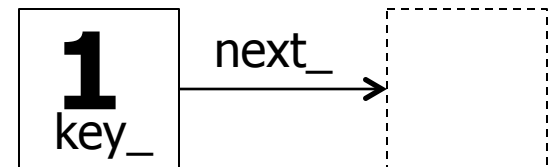
};

class Node {
...
private:
    int key_;
    Node* next_;
};
```

# Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```
class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
    ...
private:
    Node* head_;
    ...
};
```

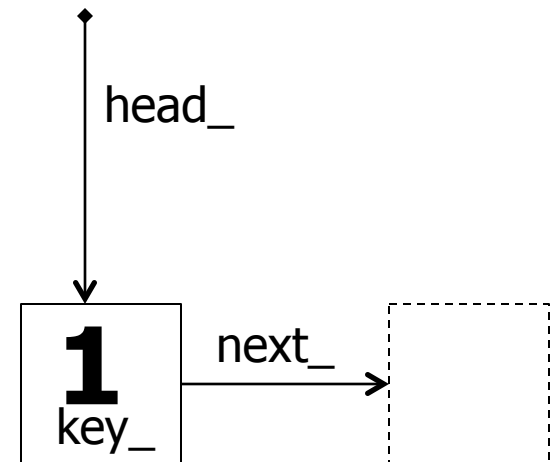
```
class Node {
...
private:
    int key_;
    Node* next_;
};
```



# Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```
class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
    ...
private:
    Node* head_;
    ...
};

class Node {
...
private:
    int key_;
    Node* next_;
};
```

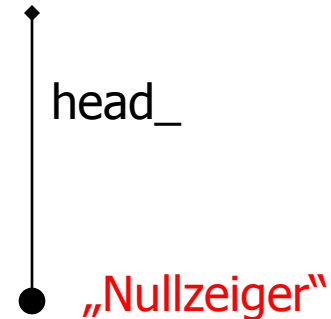




# Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```
class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
    ...
private:
    Node* head_;
    ...
};
```

```
List::List()
    : head_(0)
    {}
```



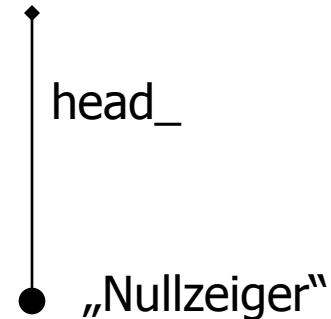
# Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```
class List {  
public:  
    // default constructor:  
    // POST: *this is an empty list  
    List();  
    ...  
private:  
    Node* head_;  
    ...  
};
```

List.h

```
List::List()  
    : head_(0)  
{}
```

List.cpp





# Eine Klasse für Listen: Einfügen neuer Elemente

---

- Zu Beginn ist die Liste leer, wie “besorgen” wir Speicher für Elemente, die wir einfügen wollen?



# Eine Klasse für Listen: Einfügen neuer Elemente

---

- Zu Beginn ist die Liste leer, wie “besorgen” wir Speicher für Elemente, die wir einfügen wollen?
- Lösung: **dynamische Speicher-  
verwaltung** mit **new** und **delete**



# New-Ausdrücke

---

**new  $T$**

Ausdruck vom Typ  $T^*$  (Zeiger)

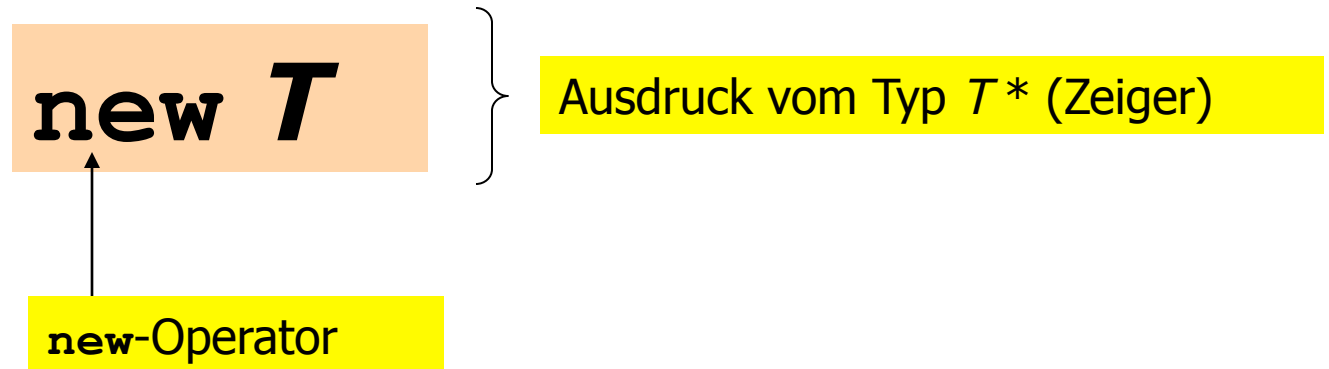
new-Operator





# New-Ausdrücke

---



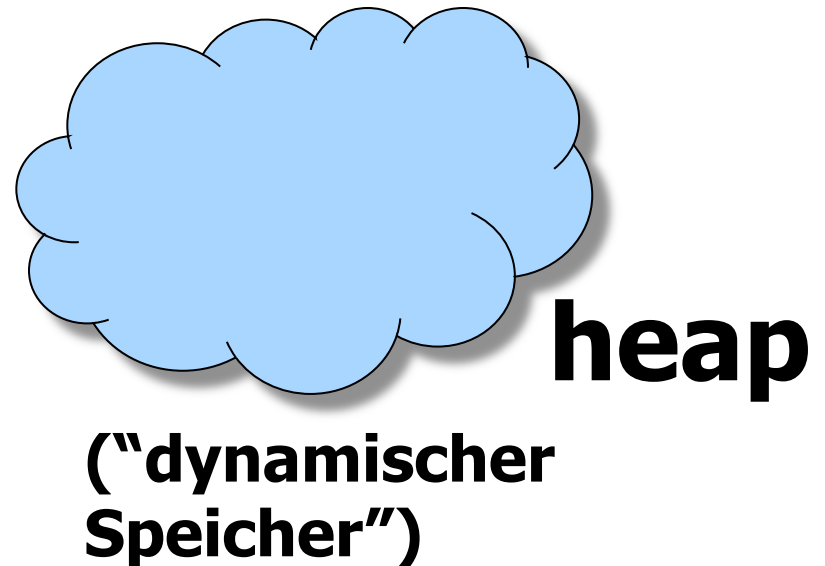
- Effekt: neuer Speicher für ein Objekt vom Typ  $T$  wird bereitgestellt; der Wert des Ausdrucks ist dessen Adresse



# Der *Heap*

---

- Hauptspeicherbereich, aus dem das Programm neuen Speicher "holen" kann.



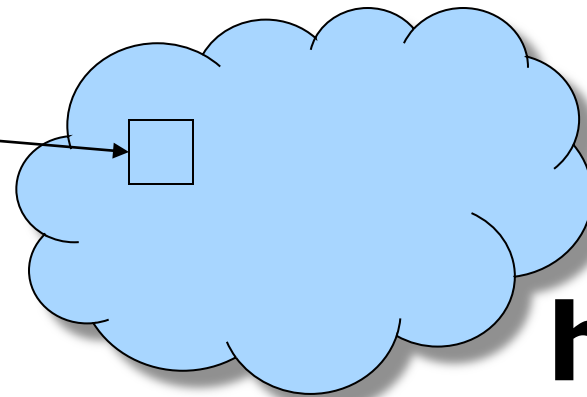


# Der *Heap*

---

- Hauptspeicherbereich, aus dem das Programm neuen Speicher "holen" kann.

```
int* i = new int;
```



**heap**

**("dynamischer  
Speicher")**



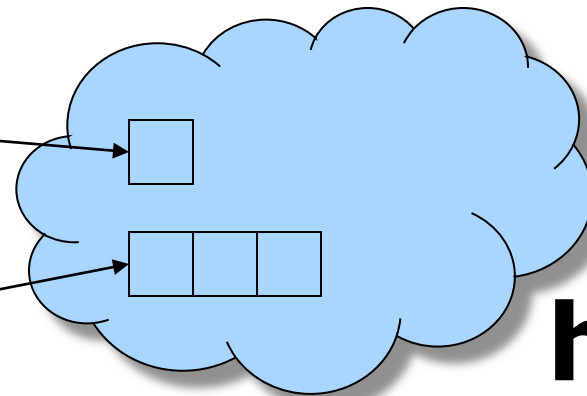
# Der *Heap*

- Hauptspeicherbereich, aus dem das Programm neuen Speicher "holen" kann.

```
int* i = new int;
```

```
int* a = new int[3];
```

Dynamisches Feld



**heap**

("dynamischer  
Speicher")



# Delete-Ausdrücke

---

Mit **new** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden:

**delete** *expr*

delete-Operator

Zeiger vom Typ  $T^*$ , der auf ein vorher mit **new** bereitgestelltes Objekt zeigt; Effekt: Speicher auf dem Heap wird wieder freigegeben.



# Delete-Ausdrücke

---

Mit **new** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden:

**delete *expr***

delete-Operator

Zeiger vom Typ  $T^*$ , der auf ein vorher mit **new** bereitgestelltes Objekt zeigt; Effekt: Speicher auf dem Heap wird wieder freigegeben.

```
int* i = new int;  
  
...  
  
delete i;
```



# Delete-Ausdrücke

---

Mit **new** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden.

**delete**[] *expr*

↑  
delete-Operator

↑  
Zeiger vom Typ  $T^*$ , der auf ein  
vorher mit **new** bereitgestelltes Feld  
zeigt; Effekt: Speicher auf dem  
Heap wird wieder freigegeben.



# Delete-Ausdrücke

---

Mit **new** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden.

**delete[]** *expr*

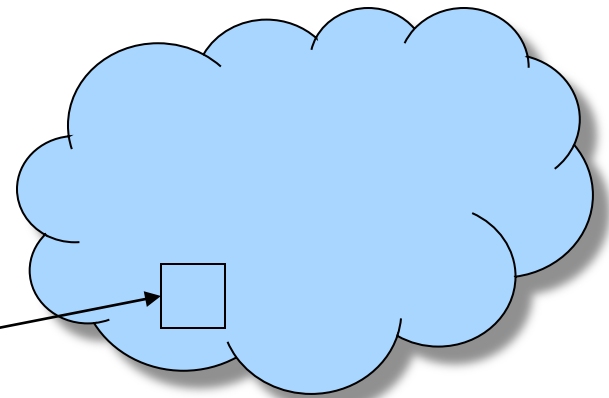
delete-Operator

```
int* a = new int[3];  
...  
delete[] a;
```

Zeiger vom Typ  $T^*$ , der auf ein vorher mit **new** bereitgestelltes Feld zeigt; Effekt: Speicher auf dem Heap wird wieder freigegeben.

# Dynamische-Speicher- Richtlinie

**Zu jedem `new`  
gibt es ein pas-  
sendes `delete`**



```
int* i = new int;
```

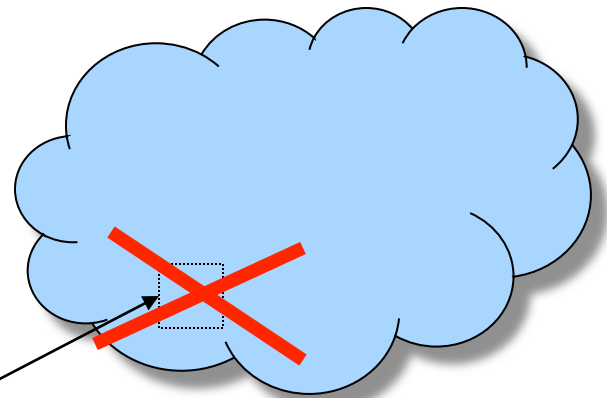
# Dynamische-Speicher- Richtlinie

**Zu jedem `new`  
gibt es ein pas-  
sendes `delete`**

```
int* i = new int;
```

```
...
```

```
delete i;
```



# Dynamische-Speicher- Richtlinie

**Zu jedem `new`  
gibt es ein pas-  
sendes `delete`**

Nichtbefolgung führt zu  
*Speicherlecks* (ungenutzter,  
aber nicht mehr verfügbarer  
Speicher auf dem Heap)







# Eine Klasse für Listen: Vorne einfügen und Ausgabe

---

## ■ Gewünschter Anwendungscode:

```
#include "List.h"
#include<iostream>

int main() {
    List l1;
    l1.push_front (1);
    l1.push_front (3);
    l1.push_front (2);
    std::cout << l1 << std::endl;
```



# Eine Klasse für Listen: Vorne einfügen und Ausgabe

---

## ■ Gewünschter Anwendungscode:

```
#include "List.h"
#include<iostream>

int main() {
    List l1;
    l1.push_front (1);           // l1 == 1
    l1.push_front (3);           // l1 == 3 1
    l1.push_front (2);           // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1
}
```

# Eine Klasse für Listen: Vorne einfügen und Ausgabe

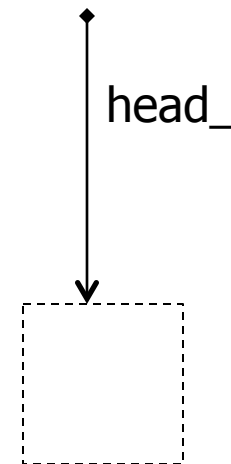
## ■ Vorne einfügen:

List.h

```
// POST: key was added before first  
//      element of *this  
void push_front(int key);
```

List.cpp

```
void List::push_front (int key)  
{  
    head_ = new Node (key, head_);  
}
```



# Eine Klasse für Listen: Vorne einfügen und Ausgabe

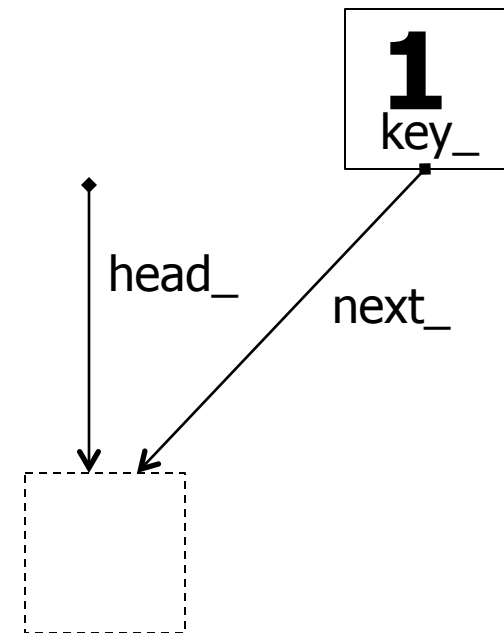
## ■ Vorne einfügen:

List.h

```
// POST: key was added before first  
//       element of *this  
void push_front(int key);
```

List.cpp

```
void List::push_front (int key)  
{  
    head_ = new Node (key, head_);  
}
```



# Eine Klasse für Listen: Vorne einfügen und Ausgabe

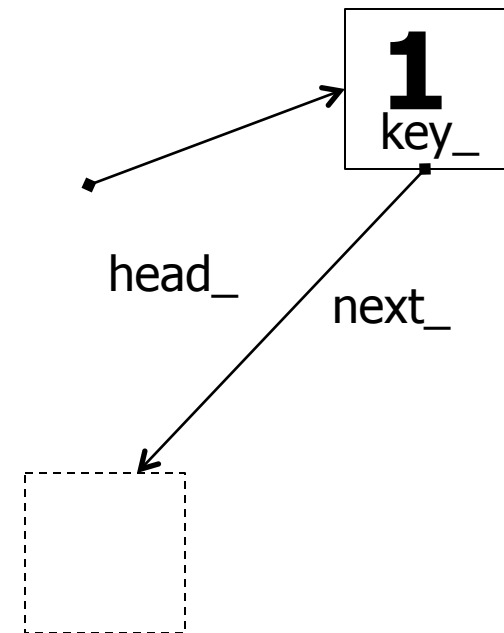
## ■ Vorne einfügen:

List.h

```
// POST: key was added before first  
//       element of *this  
void push_front(int key);
```

List.cpp

```
void List::push_front (int key)  
{  
    head_ = new Node (key, head_);  
}
```



# Eine Klasse für Listen: Vorne einfügen und Ausgabe

## ■ Vorne einfügen:

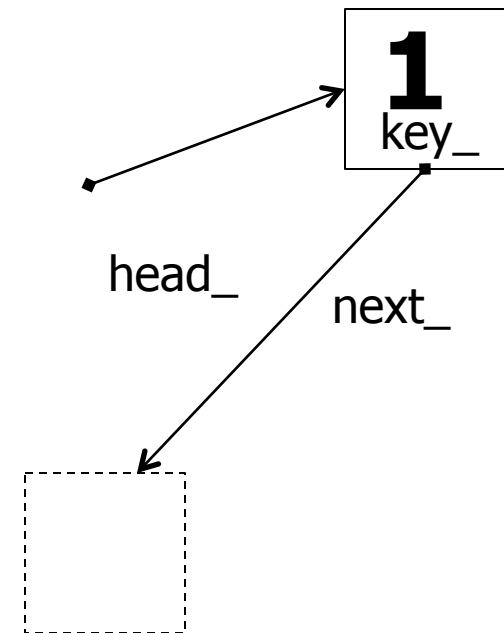
```
// Constructor
```

Node.cpp

```
Node::Node (int key, Node* next)
    : key_(key), next_(next)
{}
```

List.cpp

```
void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
```

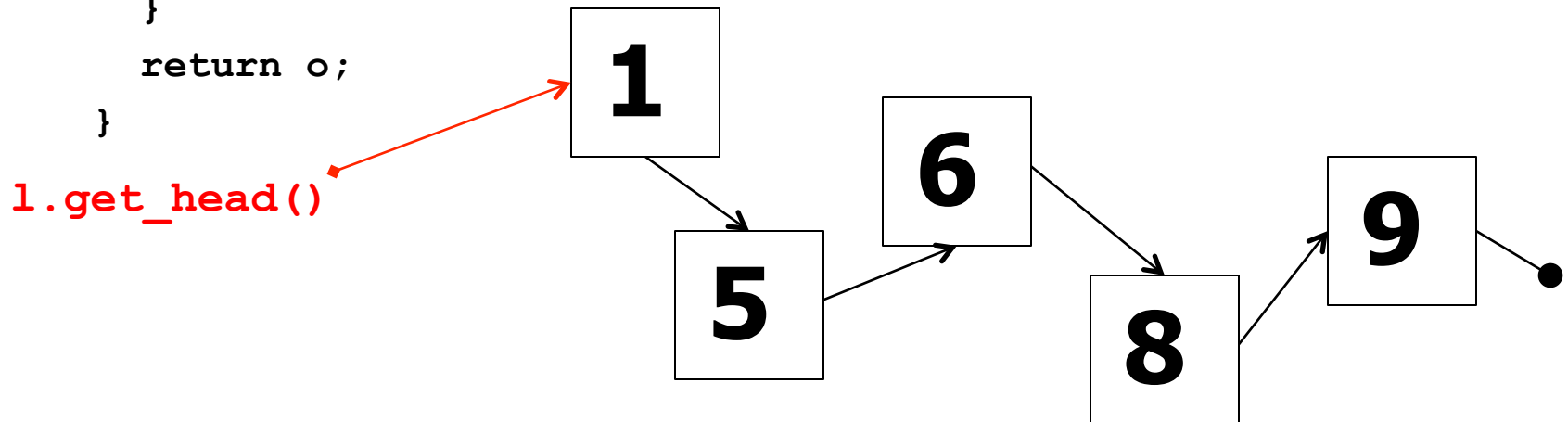


# Eine Klasse für Listen: Vorne einfügen und Ausgabe

## ■ Ausgabe:

List.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
    const Node* p = l.get_head();
    while (p != 0) {
        o << p->get_key() << " ";
        p = p->get_next();
    }
    return o;
}
```

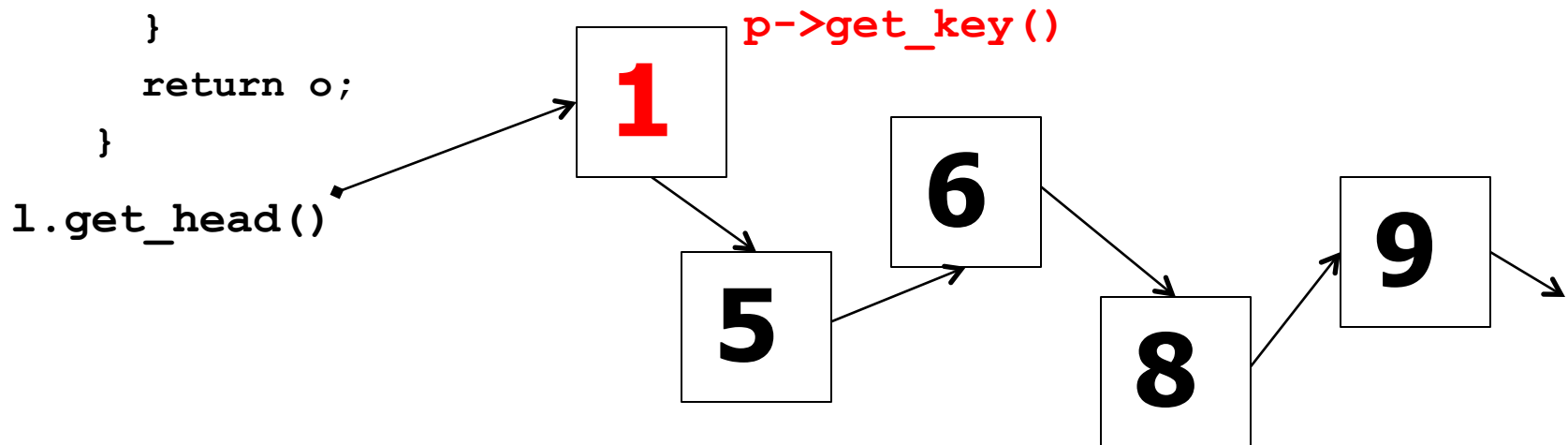


# Eine Klasse für Listen: Vorne einfügen und Ausgabe

## ■ Ausgabe:

List.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
    const Node* p = l.get_head();
    while (p != 0) {
        o << p->get_key() << " ";
        p = p->get_next();
    }
    return o;
}
```





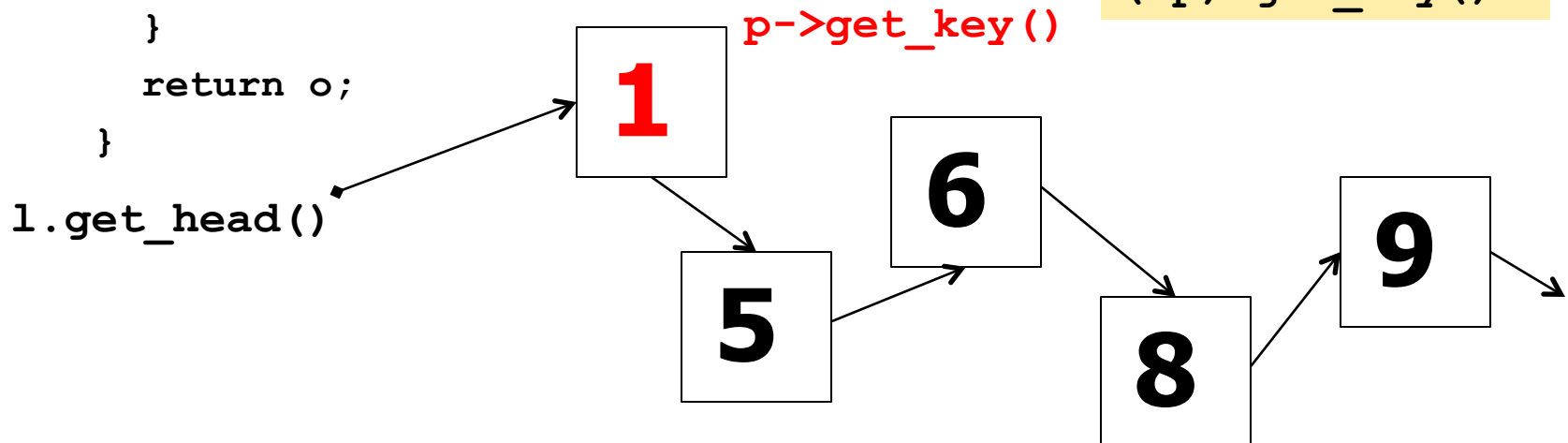
# Eine Klasse für Listen: Vorne einfügen und Ausgabe

## ■ Ausgabe:

List.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
    const Node* p = l.get_head();
    while (p != 0) {
        o << p->get_key() << " ";
        p = p->get_next();
    }
    return o;
}
```

Abkürzung für  
(\*p).get\_key()

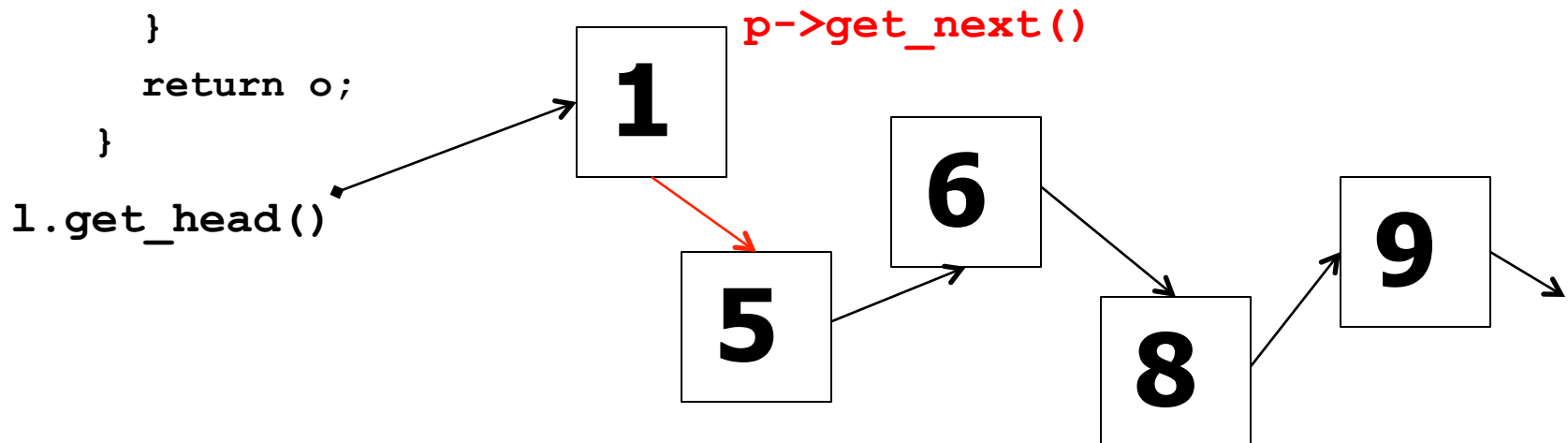


# Eine Klasse für Listen: Vorne einfügen und Ausgabe

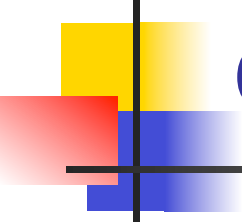
## ■ Ausgabe:

List.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
    const Node* p = l.get_head();
    while (p != 0) {
        o << p->get_key() << " ";
        p = p->get_next();
    }
    return o;
}
```



# Eine Klasse für Listen: Hinten einfügen



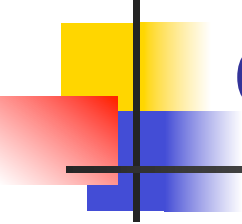
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp

# Eine Klasse für Listen: Hinten einfügen



```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp



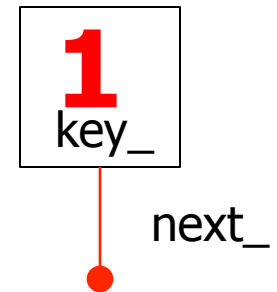
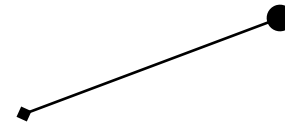
# Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last element of *this
void push_back (int key);
```

**List.h**

```
void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
```

**List.cpp**



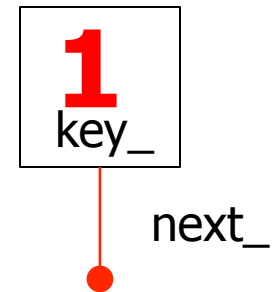
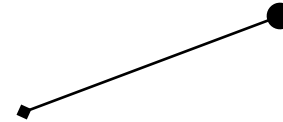
# Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp



Node.h

// Constructor

Node(int key, Node\* next = 0);

Zweites Argument  
ist 0 per Default

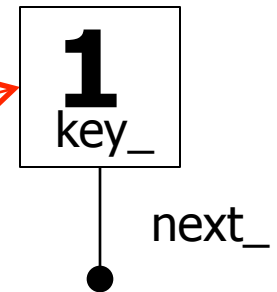
# Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp



# Eine Klasse für Listen: Hinten einfügen

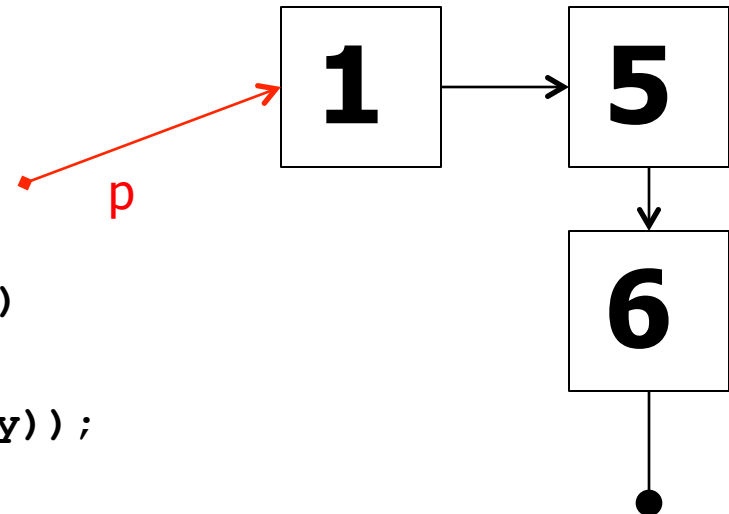
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```





# Eine Klasse für Listen: Hinten einfügen

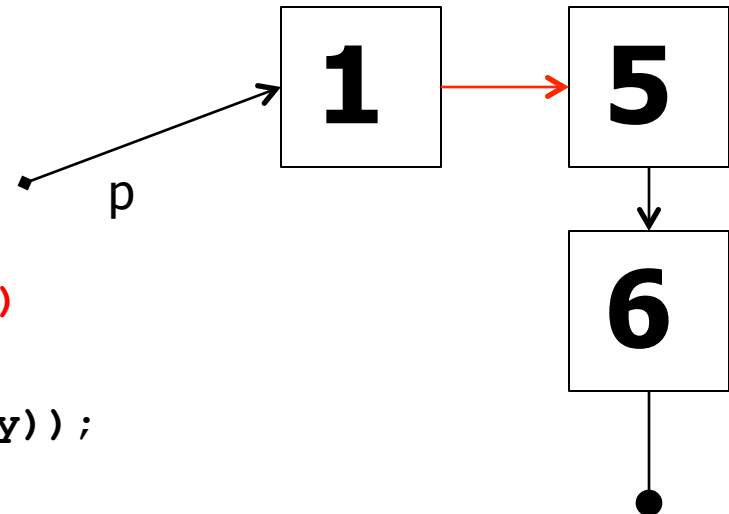
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```



# Eine Klasse für Listen: Hinten einfügen

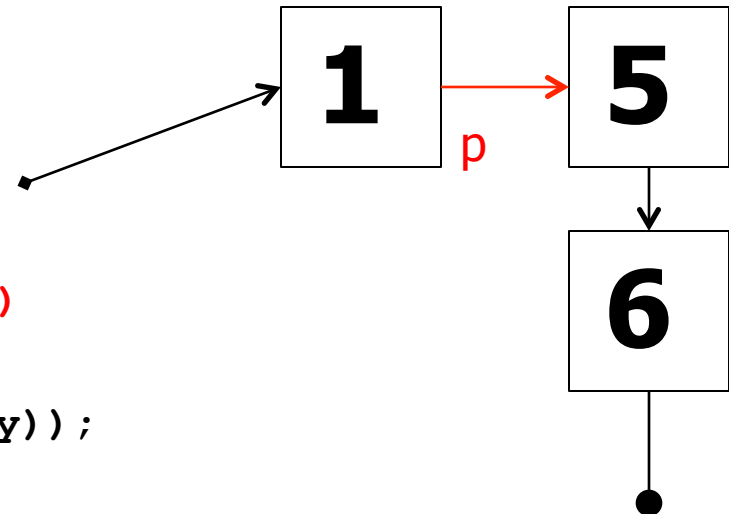
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```



# Eine Klasse für Listen: Hinten einfügen

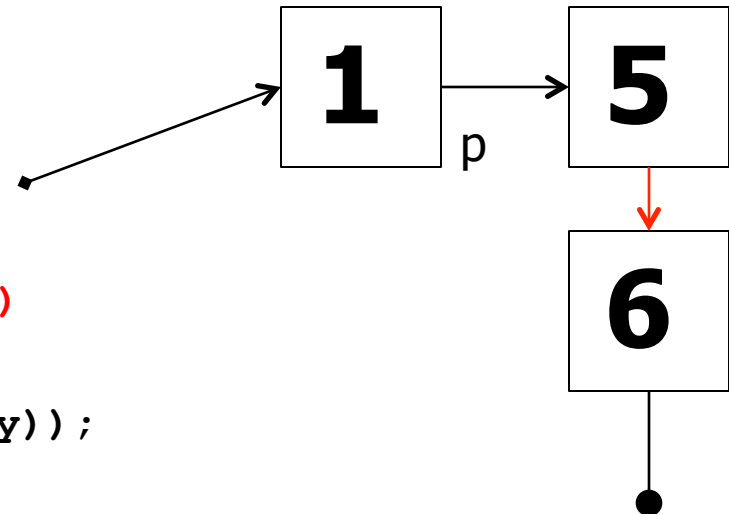
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```



# Eine Klasse für Listen: Hinten einfügen

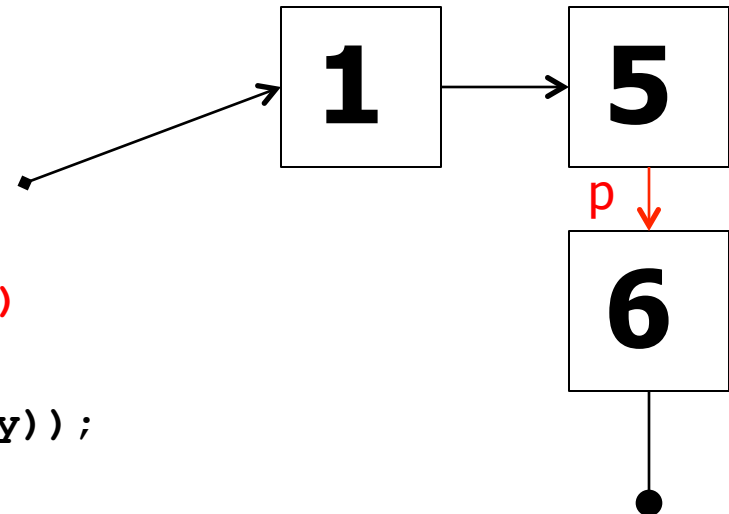
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```



# Eine Klasse für Listen: Hinten einfügen

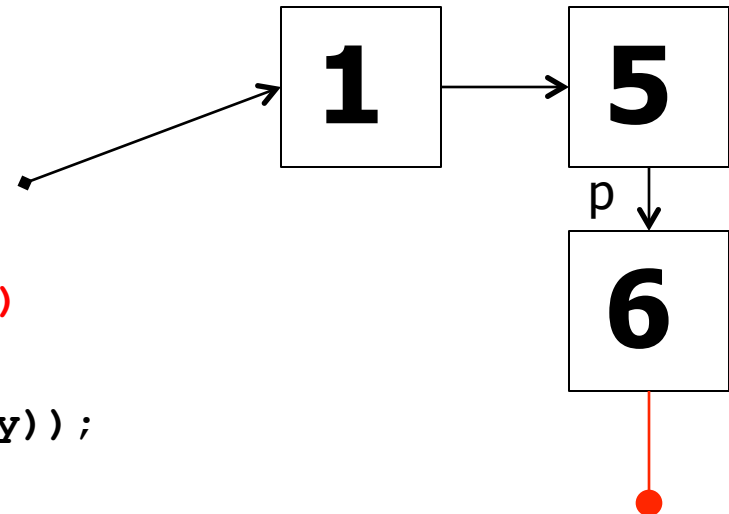
```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)
```

List.cpp

```
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```



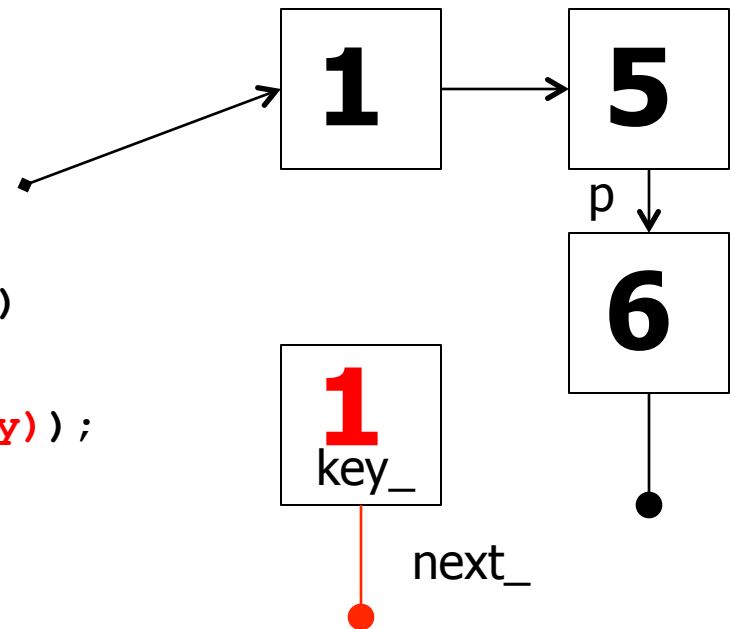
# Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp



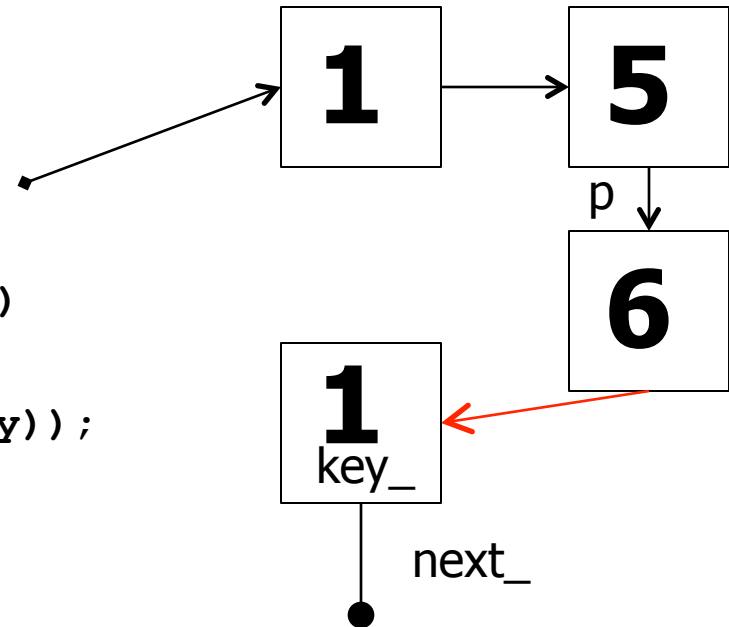
# Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last  
// element of *this  
void push_back (int key);
```

List.h

```
void List::push_back (int key)  
{  
    if (head_ == 0)  
        head_ = new Node (key);  
    else {  
        Node* p = head_;  
        while (p->get_next() != 0)  
            p = p->get_next();  
        p->set_next (new Node (key));  
    }  
}
```

List.cpp





# Eine Klasse für Listen: Der Copy-Konstruktor

---

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);           // l1 == 1  
    l1.push_front (3);           // l1 == 3 1  
    l1.push_front (2);           // l1 == 2 3 1  
    std::cout << l1 << std::endl; // 2 3 1  
  
    List l2 = l1;  
    std::cout << l2 << std::endl;  
    l1.push_back(5);  
    l1.push_back(6);  
    std::cout << l2 << std::endl;  
}
```





# Eine Klasse für Listen: Der Copy-Konstruktor

---

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);           // l1 == 1  
    l1.push_front (3);           // l1 == 3 1  
    l1.push_front (2);           // l1 == 2 3 1  
    std::cout << l1 << std::endl; // 2 3 1  
  
    List l2 = l1;                // Kopiere l1 nach l2  
    std::cout << l2 << std::endl;  
    l1.push_back(5);  
    l1.push_back(6);  
    std::cout << l2 << std::endl;  
}
```



# Eine Klasse für Listen: Der Copy-Konstruktor

---

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);           // l1 == 1  
    l1.push_front (3);           // l1 == 3 1  
    l1.push_front (2);           // l1 == 2 3 1  
    std::cout << l1 << std::endl; // 2 3 1  
  
    List l2 = l1;                // Kopiere l1 nach l2  
    std::cout << l2 << std::endl; // 2 3 1  
    l1.push_back(5);  
    l1.push_back(6);  
    std::cout << l2 << std::endl;  
}
```



# Eine Klasse für Listen: Der Copy-Konstruktor

---

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);           // l1 == 1  
    l1.push_front (3);           // l1 == 3 1  
    l1.push_front (2);           // l1 == 2 3 1  
    std::cout << l1 << std::endl; // 2 3 1  
  
    List l2 = l1;                // Kopiere l1 nach l2  
    std::cout << l2 << std::endl; // 2 3 1  
    l1.push_back(5);              // l1 == 2 3 1 5  
    l1.push_back(6);              // l1 == 2 3 1 5 6  
    std::cout << l2 << std::endl;  
}
```



# Eine Klasse für Listen: Der Copy-Konstruktor

---

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);           // l1 == 1  
    l1.push_front (3);           // l1 == 3 1  
    l1.push_front (2);           // l1 == 2 3 1  
    std::cout << l1 << std::endl; // 2 3 1  
  
    List l2 = l1;                // Kopiere l1 nach l2  
    std::cout << l2 << std::endl; // 2 3 1  
    l1.push_back(5);             // l1 == 2 3 1 5  
    l1.push_back(6);             // l1 == 2 3 1 5 6  
    std::cout << l2 << std::endl; // 2 3 1 5 6  
}
```



# Eine Klasse für Listen: Der Copy-Konstruktor

## ■ Was passiert hier?

```
int main() {  
    List l1;  
    l1.push_front (1);  
    l1.push_front (3);  
    l1.push_front (2);  
    std::cout << l1 << std::endl;  
  
    List l2 = l1;  
    std::cout << l2 << std::endl;  
    l1.push_back(5);  
    l1.push_back(6);  
    std::cout << l2 << std::endl;  
}
```

l1 wurde verändert, aber die gleichen Änderungen passieren auch mit l2. Warum?

```
// l1 == 1
```

```
// l1 == 3 1
```

```
// l1 == 2 3 1
```

```
// 2 3 1
```

```
// Kopiere l1 nach l2
```

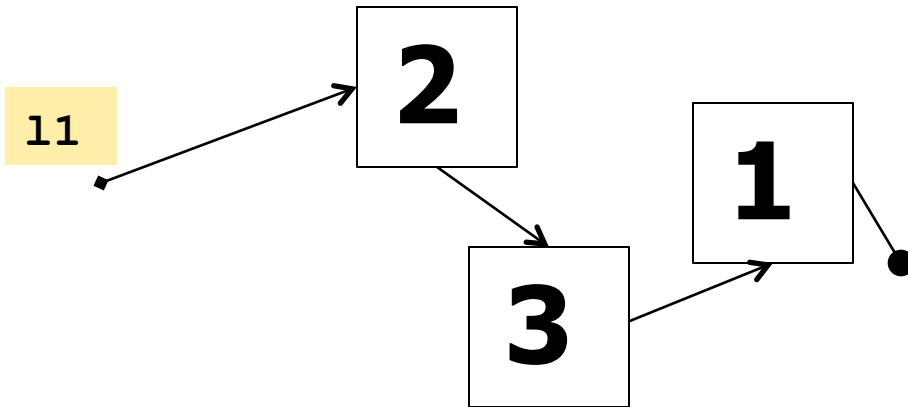
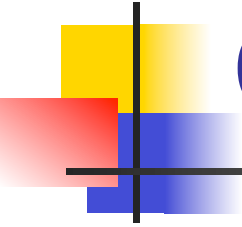
```
// 2 3 1
```

```
// l1 == 2 3 1 5
```

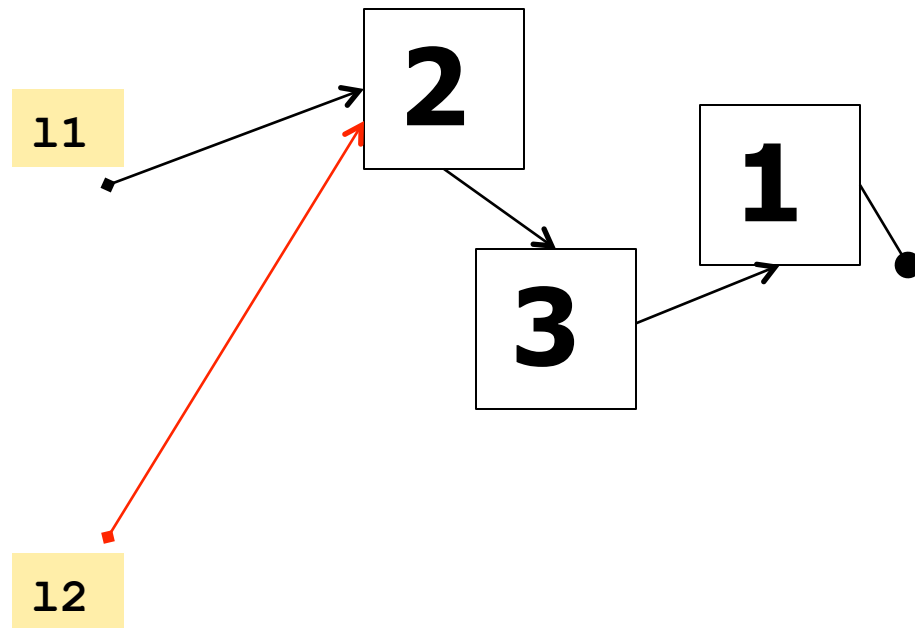
```
// l1 == 2 3 1 5 6
```

```
// 2 3 1 5 6
```

# Eine Klasse für Listen: Der Copy-Konstruktor



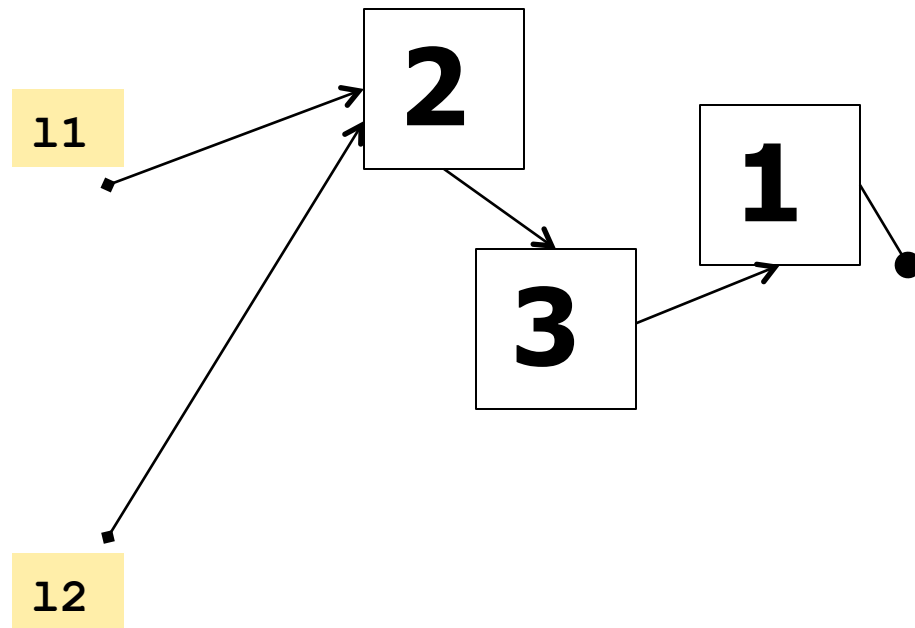
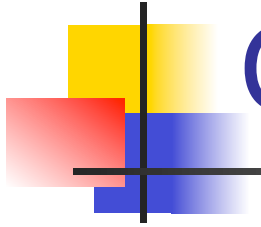
# Eine Klasse für Listen: Der Copy-Konstruktor



`List l2 = l1;`

Initialisierung erfolgt mitgliedersweise, also `l2.head_ = l1.head_` (nicht die Liste wird kopiert, sondern nur der Zeiger auf das erste Element; 11 und 12 teilen sich nun die Liste).

# Eine Klasse für Listen: Der Copy-Konstruktor

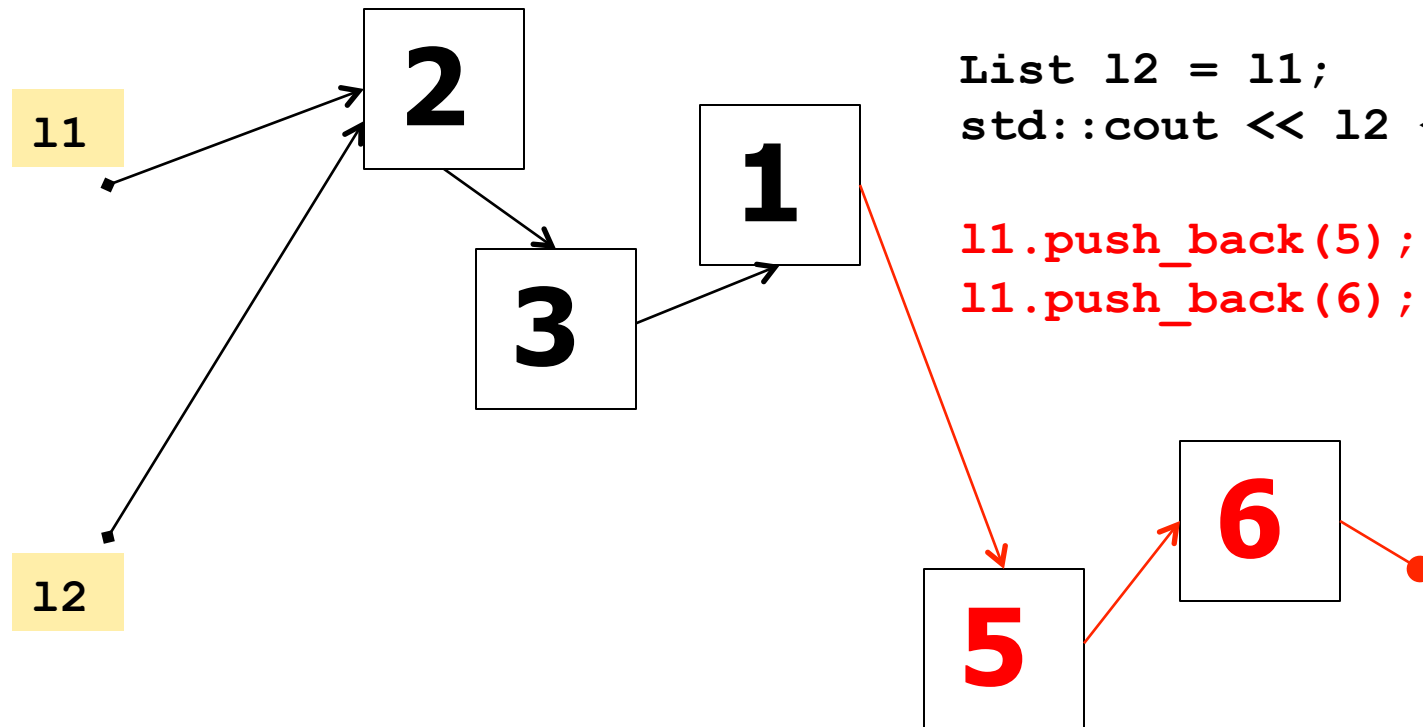
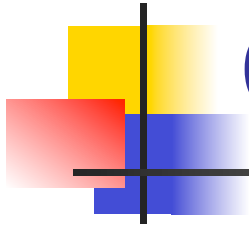


```
List l2 = l1;  
std::cout << l2 << std::endl;
```

Ausgabe 2 3 1



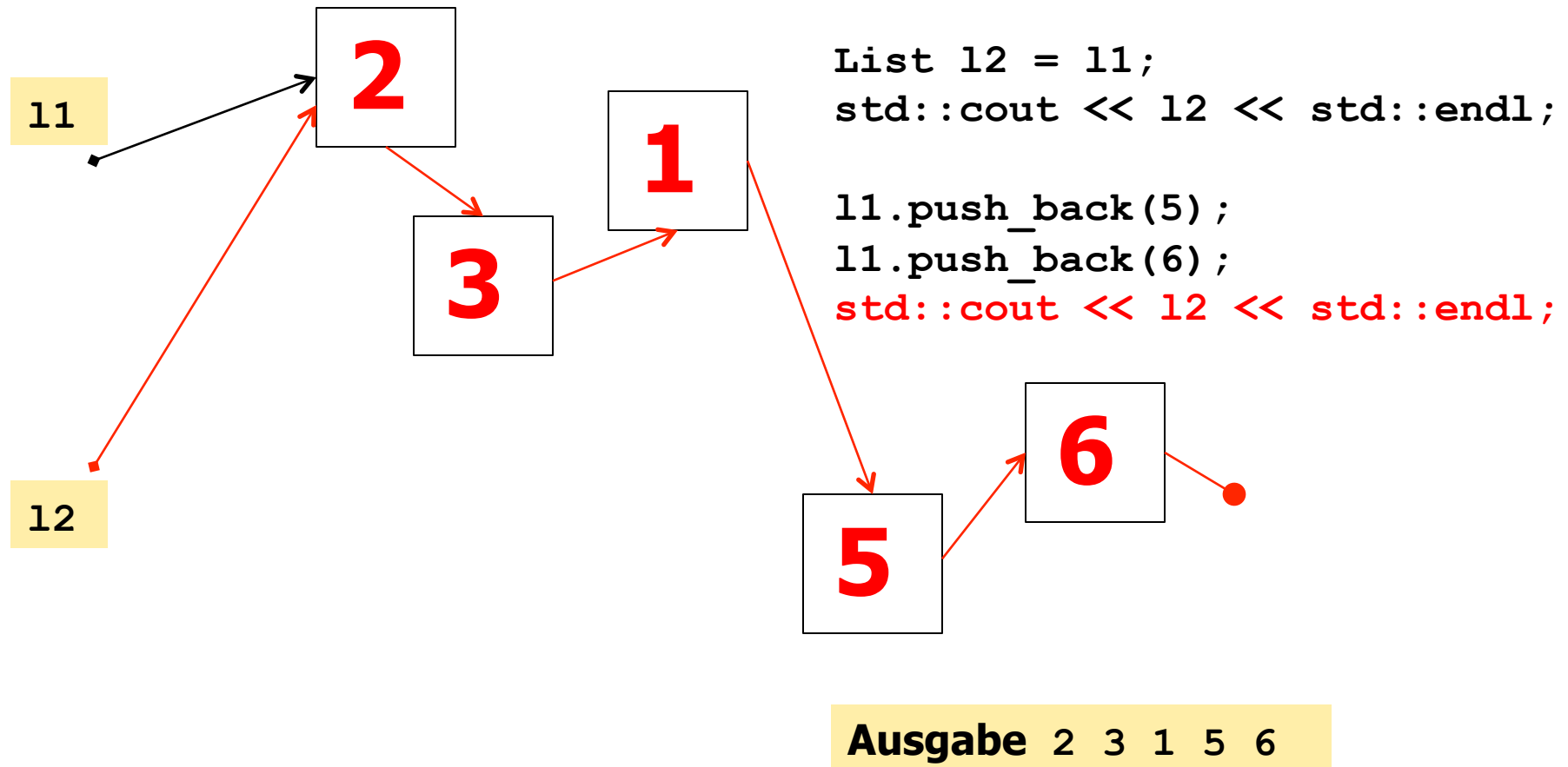
# Eine Klasse für Listen: Der Copy-Konstruktor



```
List l2 = l1;  
std::cout << l2 << std::endl;
```

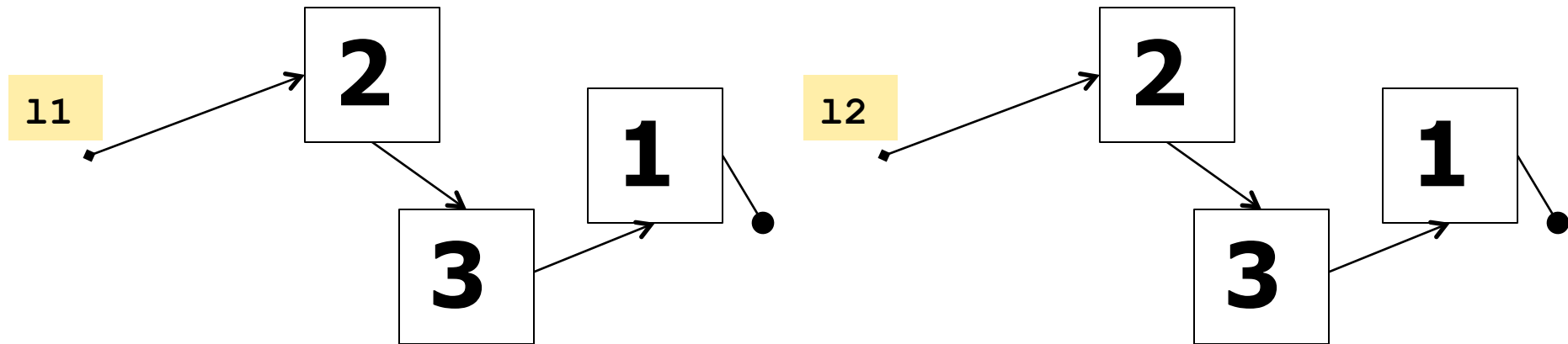
```
l1.push_back(5);  
l1.push_back(6);
```

# Eine Klasse für Listen: Der Copy-Konstruktor



# Eine Klasse für Listen: Der Copy-Konstruktor

- Wir wollen eine „echte“ Kopie, wenn wir `List 12 = 11` schreiben!





# Eine Klasse für Listen: Der Copy-Konstruktor

---

- ...der Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

$T$  (const  $T\&$   $x$ ) ;



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- ...der Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

$T$  (`const  $T\&$   $x$` ) ;

- wird automatisch aufgerufen, wenn Werte vom Typ  $T$  mit Werten vom Typ  $T$  *initialisiert* werden



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- ...der Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

$T$  (`const  $T\&$   $x$` ) ;

- wird automatisch aufgerufen, wenn Werte vom Typ  $T$  mit Werten vom Typ  $T$  *initialisiert* werden
  - `$Tx = t$ ; //  $t$  vom Typ  $T$`



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- ...der Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

$T$  (`const T& x`) ;

- wird automatisch aufgerufen, wenn Werte vom Typ  $T$  mit Werten vom Typ  $T$  *initialisiert* werden
  - $Tx$  (`t`) ; // `t` vom Typ  $T$



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- ...der Klasse  $T$  ist der eindeutige Konstruktor mit Deklaration

$T$  (**const**  $T\&$   $x$ ) ;

- wird automatisch aufgerufen, wenn Werte vom Typ  $T$  mit Werten vom Typ  $T$  *initialisiert* werden
  - Initialisierung von formalen Funktions-Argumenten und Rückgabewerten





# Eine Klasse für Listen: Der Copy-Konstruktor

---

```
// copy constructor
```

List.cpp

```
List::List (const List& l)
    : head_ (0)
{
    copy (l.head_);
}
```

```
public:
```

List.h

```
List (const List& l);
```

```
...
```

```
private:
```

```
// PRE: *this is empty
```

```
// POST: list starting at from was copied to *this
```

```
void copy (const Node* from);
```

# Eine Klasse für Listen: Der Copy-Konstruktor

```
// copy constructor
List::List (const List& l)
    : head_ (0)
{
    copy (l.head_);
}
```

List.cpp

Warum ist **const List& l** hier zwingend notwendig, während **List l** nicht geht ?

public: List.h

```
List (const List& l);
```

...

private:

```
// PRE: *this is empty
```

```
// POST: list starting at from was copied to *this
```

```
void copy (const Node* from);
```

# Eine Klasse für Listen: Der Copy-Konstruktor

```
// copy constructor
List::List (const List& l)
    : head_ (0)
{
    copy (l.head_);
}
```

public: **List.h**

```
List (const List& l);
```

...

private:

```
// PRE: *this is empty
```

```
// POST: list starting at from was copied to *this
```

```
void copy (const Node* from);
```

**List.cpp**

Aufruf eines Copy-Konstruktors mit Deklaration **List (List l);** müsste zuerst den Copy-Konstruktor aufrufen (Initialisierung des formalen Arguments!); unendliche Rekursion!



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert mitgliedersweise)



# Eine Klasse für Listen: Der Copy-Konstruktor

---

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert mitgliedsweise)

Das ist für unsere Klasse `List` genau nicht das, was wir wollen (es wird nur der Zeiger `head_` kopiert, *nicht* jedoch die dahinterliegende Liste – wir erhalten Alias-Semantik)!



# Eine Klasse für Listen: Die (private) Copy-Funktion

---

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```

# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```

**\*this.head**

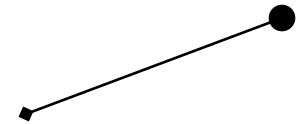


# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```

**\*this.head**



**from** →

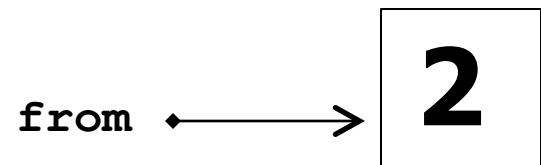
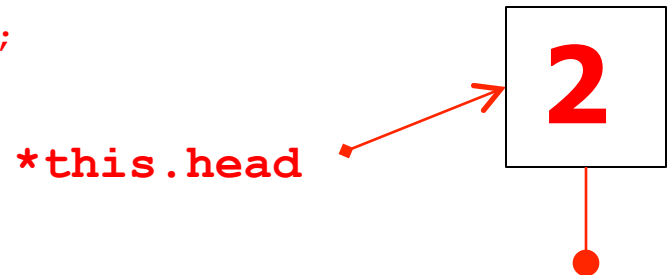
**2**



# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

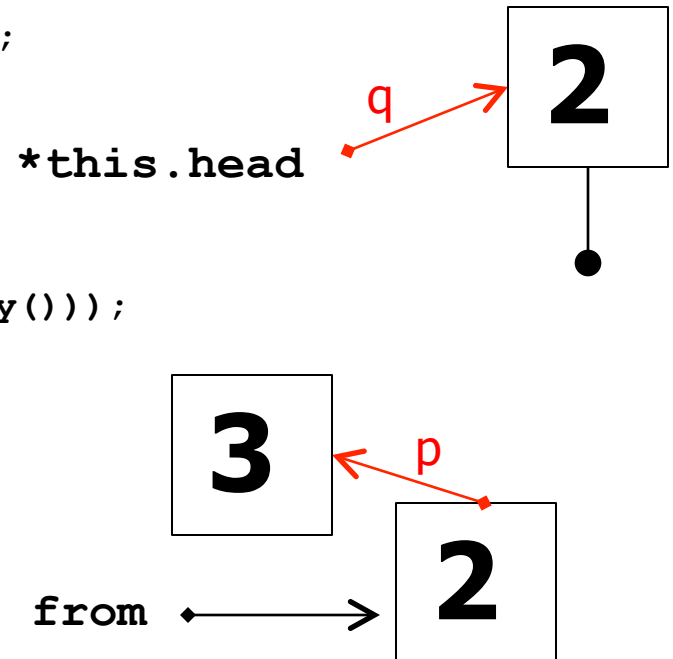
```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```



# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

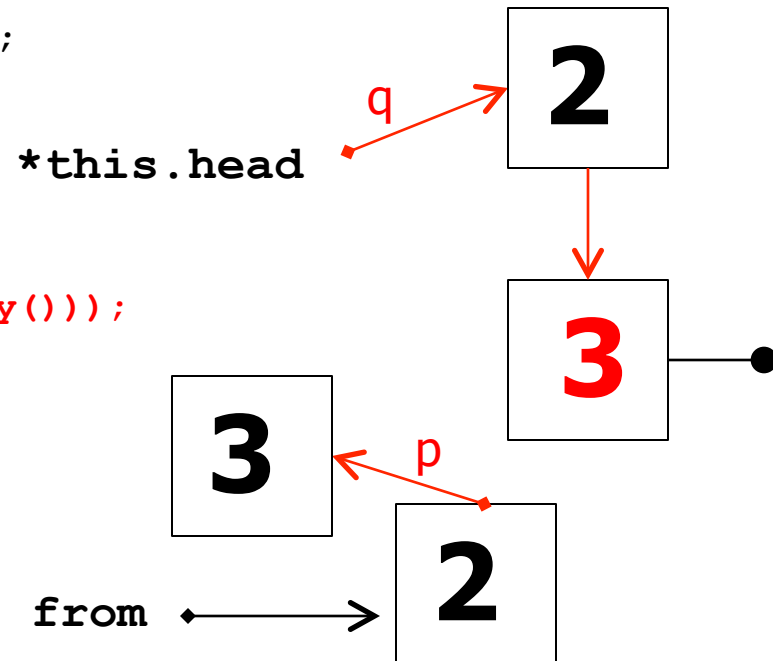
```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```



# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

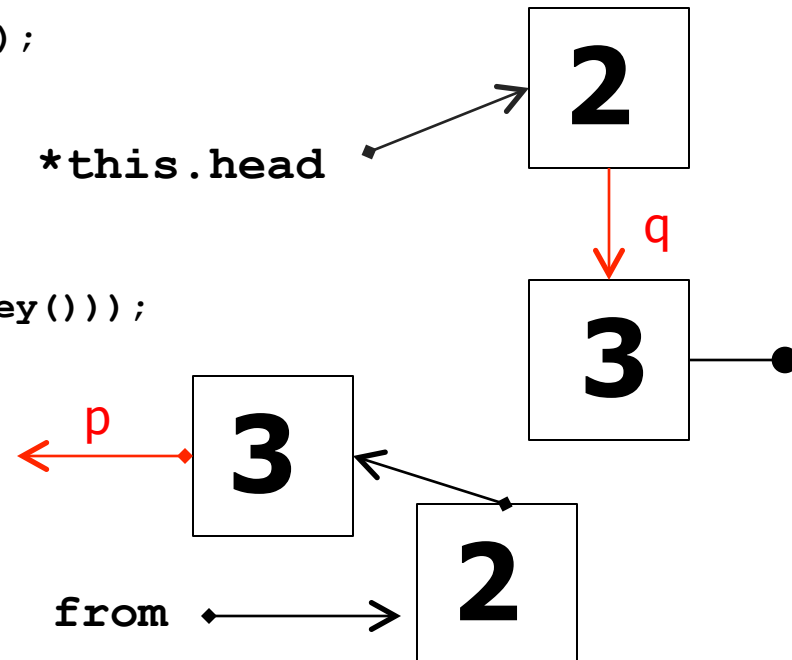
```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```



# Eine Klasse für Listen: Die (private) Copy-Funktion

```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```

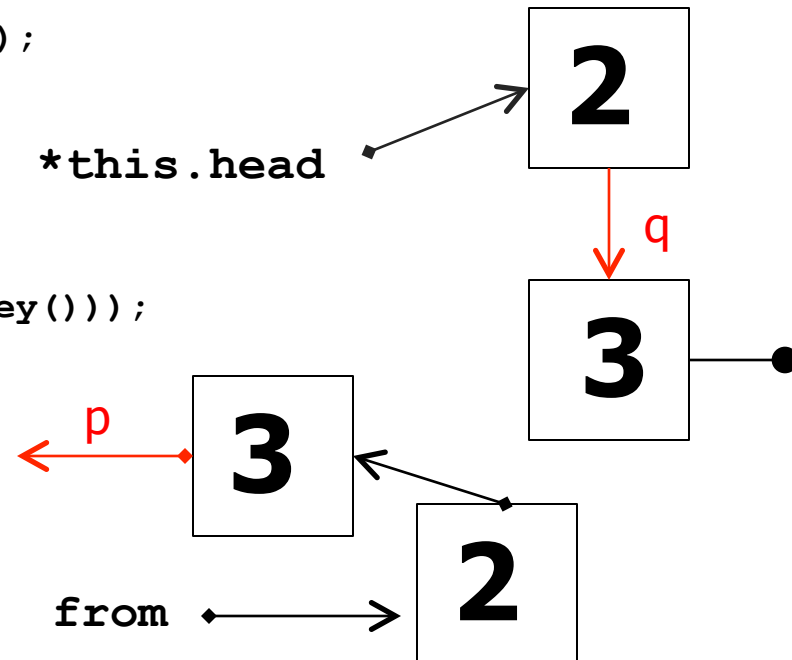


# Eine Klasse für Listen: Die (private) Copy-Funktion

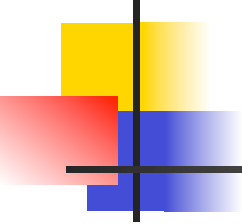
```
// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
```

und so weiter...

```
// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
```



# Eine Klasse für Listen: Echtes Kopieren



```
List l1;  
l1.push_front (1);  
l1.push_front (3);  
l1.push_front (2);  
std::cout << l1 << std::endl; // 2 3 1
```

```
List l2 = l1; // Aufruf „unseres“ Copy-Konstruktors  
std::cout << l2 << std::endl; // 2 3 1
```

```
l1.push_back (5);  
l1.push_back (6);  
std::cout << l1 << std::endl; // 2 3 1 5 6  
std::cout << l2 << std::endl; // 2 3 1
```



# Eine Klasse für Listen: Der Zuweisungsoperator

---

- Überladung von `operator=` als Mitglieds-Funktion



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List l3;  
l3 = l2;
```

- Überladung von `operator=` als Mitglieds-Funktion





# Eine Klasse für Listen: Der Zuweisungsoperator

```
List l3;  
l3 = l2;
```

- Überladung von `operator=` als Mitglieds-Funktion
- ähnlich wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
  - Freigabe des Speichers für den “alten” Wert
  - Prüfen auf Selbstzuweisungen (`l3 = l3`), die keinen Effekt haben sollten



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this;
}
```



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this;
}
```

Prüfen auf Selbstzuweisungen



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this;
}
```

Freigabe des Speichers für den  
"alten" Wert



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this;
}
```

Das eigentliche Kopieren



# Eine Klasse für Listen: Der Zuweisungsoperator

---

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this;
}
```

Konvention: Zuweisungsoperator gibt den neuen Wert als L-Wert zurück.



# Eine Klasse für Listen: Der Zuweisungsoperator

---

- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist mitgliedsweise zu)

Das ist für unsere Klasse `List` wiederum nicht das, was wir wollen (es wird nur der Zeiger `head_` kopiert, *nicht* jedoch die dahinterliegende Liste – wir erhalten Alias-Semantik)!



# Eine Klasse für Listen: Der Zuweisungsoperator

---

...

```
std::cout << l1 << std::endl; // 2 3 1 5 6  
std::cout << l2 << std::endl; // 2 3 1
```

```
List l3;
```

```
l3 = l1;
```

Aufruf „unseres“ Zuweisungsoperators

```
std::cout << l3 << std::endl; // 2 3 1 5 6
```





# Eine Klasse für Listen: Der Destruktor

---

- Problem: Quelltext enthält jede Menge **new**'s, aber keine **delete**'s, das führt zu Speicherlecks

```
void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
```



# Eine Klasse für Listen: Der Destruktor

---

- Problem: Quelltext enthält jede Menge **new**'s, aber keine **delete**'s, das führt zu Speicherlecks
- Die **delete**'s sollen passieren, wenn wir die Liste nicht mehr brauchen.

```
void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
```



# Eine Klasse für Listen: Der Destruktor

---

- Problem: Quelltext enthält jede Menge **new's**, aber keine **delete's**, das führt zu Speicherlecks
- Die **delete's** sollen passieren, wenn wir die Liste nicht mehr brauchen....
- Nämlich genau dann, wenn ihre automatische Speicherdauer endet!

# Eine Klasse für Listen: Der Destruktor

- Nämlich genau dann, wenn ihre automatische Speicherdauer endet!

```
int main() {  
    List l1;  
    l1.push_front (1);  
    l1.push_front (3);  
    l1.push_front (2);  
    std::cout << l1 << std::endl;    // 2 3 1  
    return 0;  
} ←
```

Ziel genau wie bei fundamentalen Typen:  
Ende des Gültigkeitsbereichs, der Speicher  
wird wieder freigegeben.



# Eine Klasse für Listen: Der Destruktor

---

- Der *Destruktor* ist eine spezielle Mitgliedsfunktion, die automatisch aufgerufen wird, wenn die Speicherdauer eines Klassenobjekts endet.



# Eine Klasse für Listen: Der Destruktor

---

- Der *Destruktor* ist eine spezielle Mitgliedsfunktion, die **automatisch** aufgerufen wird, wenn die Speicherdauer eines Klassenobjekts endet.
- Wird kein Destruktor deklariert, so wird er automatisch erzeugt und ruft die Destruktoren für die Datenmitglieder auf (hier: Zeiger `head_`, kein Effekt)



# Eine Klasse für Listen: Der Destruktor

---

```
List::~~List()  
{  
    clear();  
}
```

Destruktor: Name wie die Klasse, mit ~ davor, und stets ohne Argumente



# Eine Klasse für Listen: Der Destruktor

---

```
List::~~List()  
{  
    clear();  
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
```

```
{
```

```
    clear();
```

```
}
```

```
void List::clear ()
```

```
{
```

```
    Node* p = head_;
```

```
    while (p != 0) {
```

```
        p = p->get_next();
```

```
        delete head_;
```

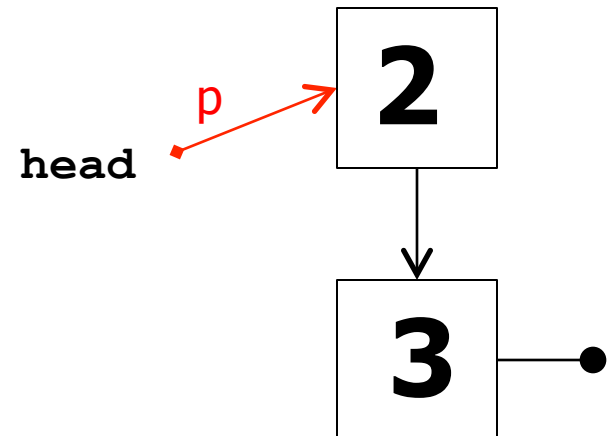
```
        head_ = p;
```

```
    }
```

```
    assert (head_ == 0);
```

```
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
```

```
{
```

```
    clear();
```

```
}
```

```
void List::clear ()
```

```
{
```

```
    Node* p = head_;
```

```
    while (p != 0) {
```

```
        p = p->get_next();
```

```
        delete head_;
```

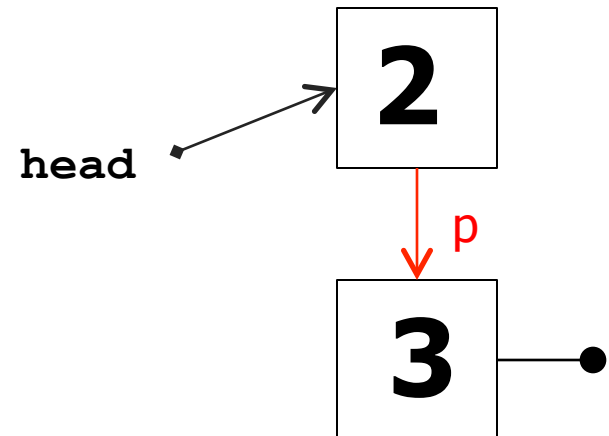
```
        head_ = p;
```

```
    }
```

```
    assert (head_ == 0);
```

```
}
```

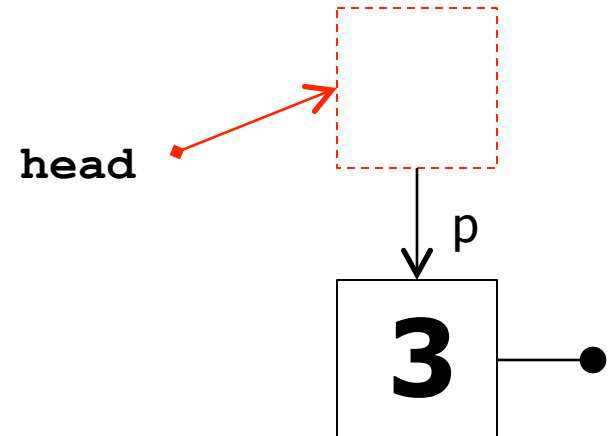
Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()  
{  
    clear();  
}  
void List::clear ()  
{  
    Node* p = head_;  
    while (p != 0) {  
        p = p->get_next();  
        delete head_;  
        head_ = p;  
    }  
    assert (head_ == 0);  
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

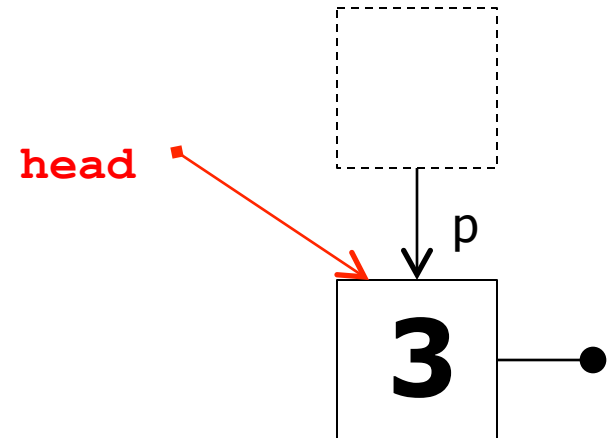


# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
{
    clear();
}

void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

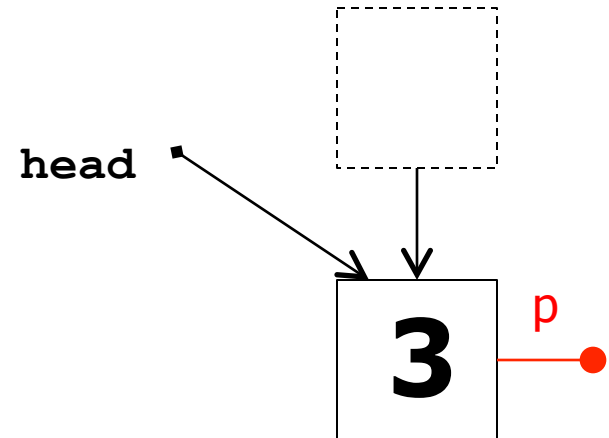


# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
{
    clear();
}

void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
```

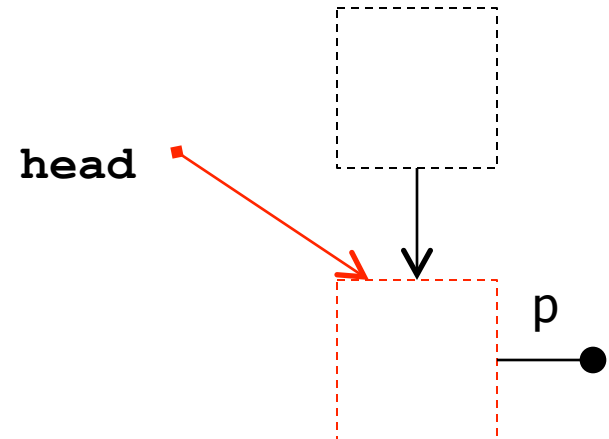
Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()  
{  
    clear();  
}  
void List::clear ()  
{  
    Node* p = head_;  
    while (p != 0) {  
        p = p->get_next();  
        delete head_;  
        head_ = p;  
    }  
    assert (head_ == 0);  
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
```

```
{
```

```
    clear();
```

```
}
```

```
void List::clear ()
```

```
{
```

```
    Node* p = head_;
```

```
    while (p != 0) {
```

```
        p = p->get_next();
```

```
        delete head_;
```

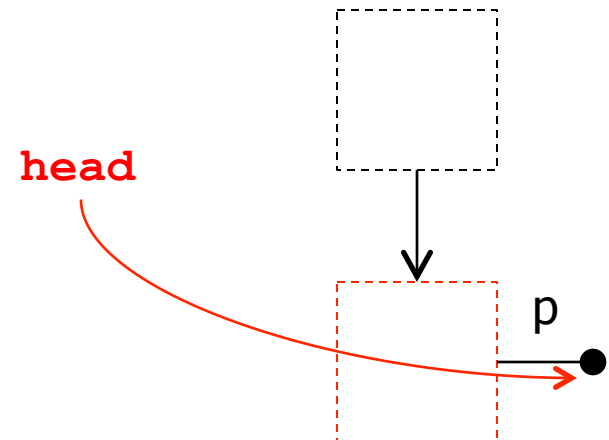
```
        head_ = p;
```

```
    }
```

```
    assert (head_ == 0);
```

```
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.



# Eine Klasse für Listen: Der Destruktor

```
List::~~List()
```

```
{
```

```
    clear();
```

```
}
```

```
void List::clear ()
```

```
{
```

```
    Node* p = head_;
```

```
    while (p != 0) {
```

```
        p = p->get_next();
```

```
        delete head_;
```

```
        head_ = p;
```

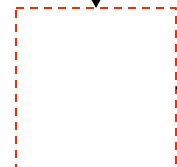
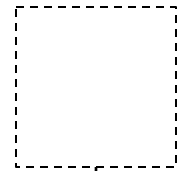
```
    }
```

```
    assert (head_ == 0);
```

```
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

head



p







# Dynamischer Datentyp

---

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Listen)



# Dynamischer Datentyp

---

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Listen)
- andere typische Anwendungen:
  - Stapel
  - Bäume
  - Graphen



# Dynamischer Datentyp

---

- sollte immer mindestens
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperatorhaben.



# Listen: Elemente entfernen

---

- Zur Erinnerung: damit haben wir Listen motiviert!



# Listen: Elemente entfernen

---

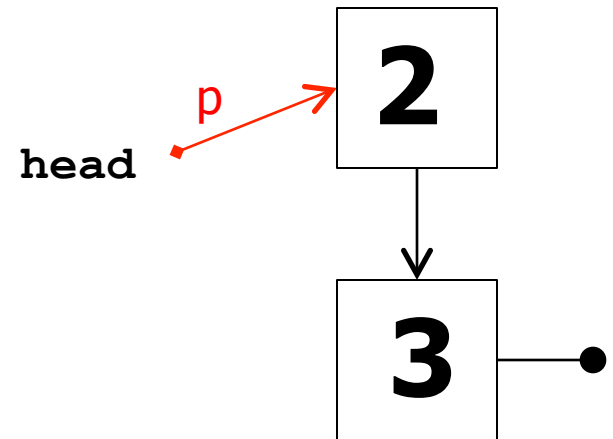
- Zur Erinnerung: damit haben wir Listen motiviert!
- Gewünschte Funktionalität:

```
// POST: the first occurrence of key was removed from *this  
//      if *this does not contain key, nothing happened  
void remove (int key);
```

# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else
                    p = q;
            }
    }
}
```

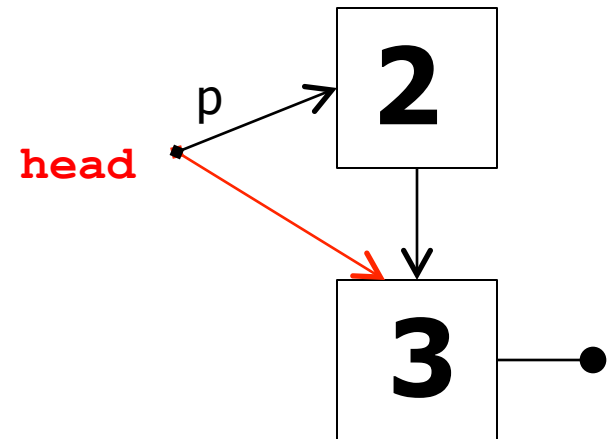
Fall 1: erstes Element wird entfernt (**key == 2**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_->get_next();
            delete p;
        } else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else
                    p = q;
            }
    }
}
```

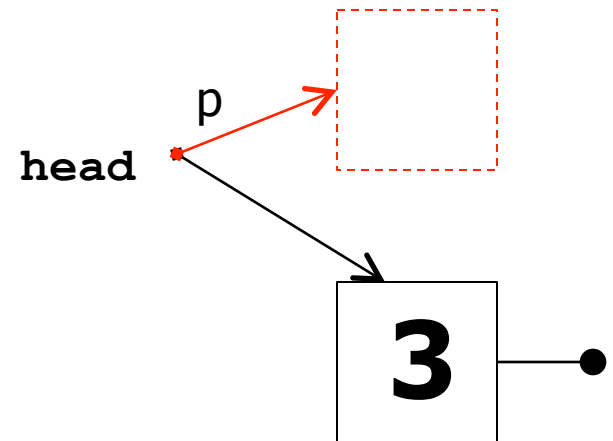
Fall 1: erstes Element wird entfernt (**key == 2**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
        while (p->get_next() != 0) {
            Node* q = p->get_next();
            if (q->get_key() == key) {
                p->set_next(q->get_next());
                delete q;
                break;
            } else
                p = q;
        }
    }
}
```

Fall 1: erstes Element wird entfernt (**key == 2**).

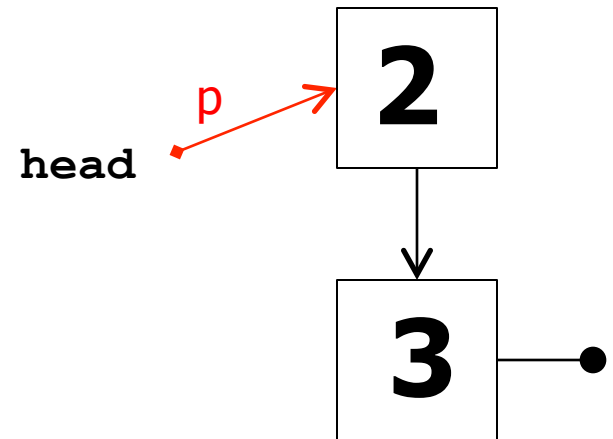




# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_->get_next();
            delete p;
        } else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else
                    p = q;
            }
    }
}
```

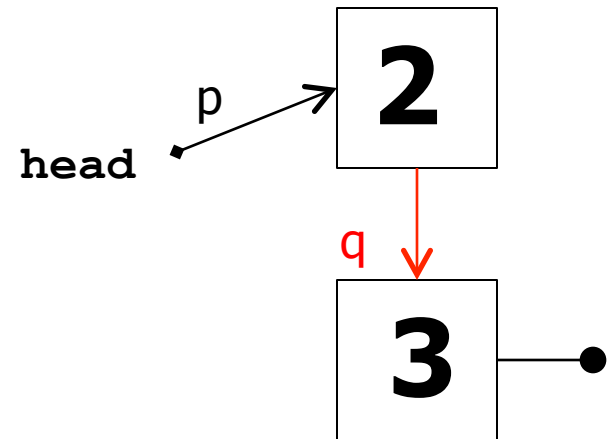
Fall 2: anderes Element  
wird entfernt (**key == 3**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else
                    p = q;
            }
    }
}
```

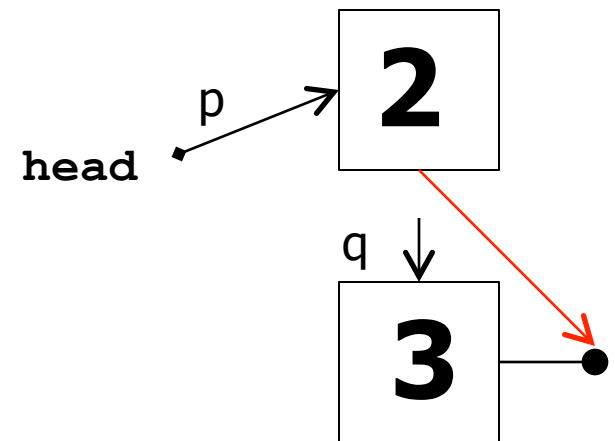
Fall 2: anderes Element  
wird entfernt (**key == 3**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
        while (p->get_next() != 0) {
            Node* q = p->get_next();
            if (q->get_key() == key) {
                p->set_next(q->get_next());
                delete q;
                break;
            } else
                p = q;
        }
    }
}
```

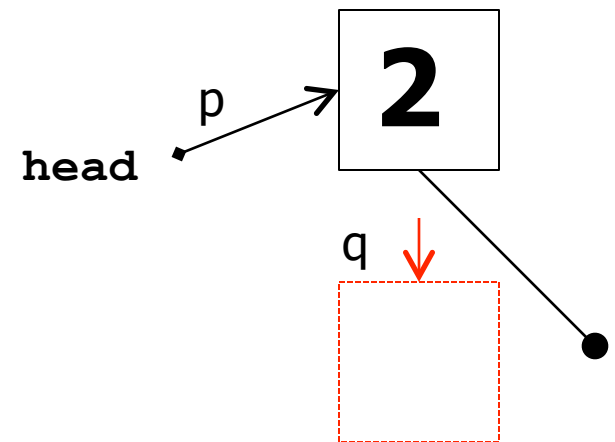
Fall 2: anderes Element  
wird entfernt (**key == 3**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
        while (p->get_next() != 0) {
            Node* q = p->get_next();
            if (q->get_key() == key) {
                p->set_next(q->get_next());
                delete q;
                break;
            } else
                p = q;
        }
    }
}
```

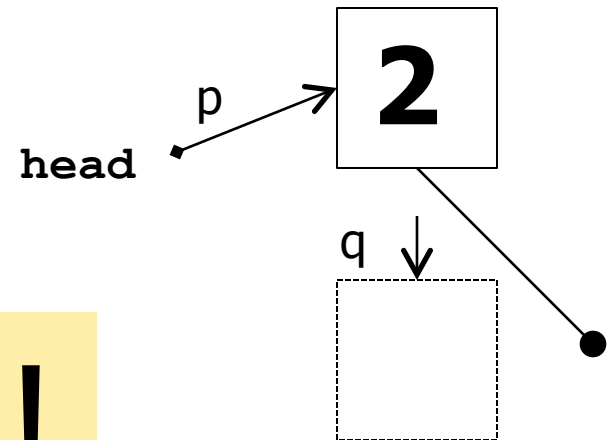
Fall 2: anderes Element  
wird entfernt (**key == 3**).



# Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        } else
        while (p->get_next() != 0) {
            Node* q = p->get_next();
            if (q->get_key() == key) {
                p->set_next(q->get_next());
                delete q;
                break;
            } else
                p = q;
        }
    }
}
```

Fall 2: anderes Element  
wird entfernt (**key == 3**).



Fertig!