Einführung in die Objektorientierte Programmierung
Vorlesung 18: Lineare Datenstrukturen

Sebastian Küpper

Unzulänglichkeit von Feldern

- Wenn ein Unternehmen alle Rechnungen eines Jahres verwalten möchte, sind Felder ungeeignet
- Grund: Die Feldgröße muss von Beginn an festgelegt werden

Unzulänglichkeit von Feldern

- Wenn ein Unternehmen alle Rechnungen eines Jahres verwalten möchte, sind Felder ungeeignet
- Grund: Die Feldgröße muss von Beginn an festgelegt werden
- Problem 1: Wählt man das Feld zu klein, kann man nicht alle Rechnungen sichern
- Problem 2: Wählt man das Feld zu groß, verschwendet man Speicherplatz

Unzulänglichkeit von Feldern

- Wenn ein Unternehmen alle Rechnungen eines Jahres verwalten möchte, sind Felder ungeeignet
- Grund: Die Feldgröße muss von Beginn an festgelegt werden
- Problem 1: Wählt man das Feld zu klein, kann man nicht alle Rechnungen sichern
- Problem 2: Wählt man das Feld zu groß, verschwendet man Speicherplatz
- Darum: Dynamische Datenstrukturen.

Eigenschaften dynamischer Datenstrukturen

Dynamische Datenstrukturen...

- können ad hoc vergrößert und verkleinert werden
- ermöglichen ein komfortables Löschen von Einträgen
- gibt es in verschiedenen Formen für verschiedene Anwendungszwecke

Lineare Datenstrukturen

- In diesem Video: Lineare Datenstrukturen
- d.h. Daten werden linear, mit eindeutigem Vorgänger und Nachfolger abgelegt
- Unterscheidung verschiedener linearer Datenstrukturen durch Zugriffsrechte

Verkettete Liste

Definition (Verkettete Liste)

- Eine einfach verkettete Liste ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält.

Verkettete Liste

Definition (Verkettete Liste)

- Eine einfach verkettete Liste ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält.
- Erster Knoten: Kopf
- Letzter Knoten: Schwanz

Verkettete Liste

Definition (Verkettete Liste)

- Eine einfach verkettete Liste ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält.
- Erster Knoten: Kopf
- Letzter Knoten: Schwanz

Klassisch rekursive Definition: Eine Liste ist entweder leer oder es ist ein Eintrag gefolgt von einer Liste

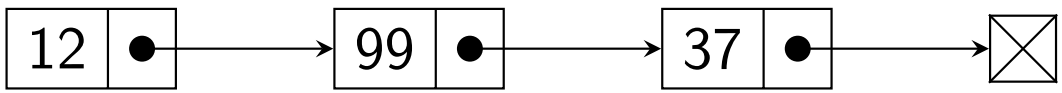
Verkettete Liste

Definition (Verkettete Liste)

- Eine einfach verkettete Liste ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält.
- Erster Knoten: Kopf
- Letzter Knoten: Schwanz

Klassisch rekursive Definition: Eine Liste ist entweder leer oder es ist ein Eintrag gefolgt von einer Liste
Achtung: Schwanz bezeichnet manchmal auch die gesamte Liste ohne Kopf

Beispiel verkettete Liste



Kopf

Schwanz

Leere verkettete Liste

Kopf=Schwanz

Implementation der verketteten Liste

```
public class LinkedList {
   private ListNode head;
   public LinkedList() {
      this.head = null;
   }
}
```

Implementation der ListNode

```
public class ListNode {
   private int entry;
   private ListNode next;
   public ListNode(int value) {
      this(value, null);
   }
   public ListNode(int value, ListNode nextNode) {
      this.entry = value;
      this.next = nextNode;
   }
   //Getter und Setter fuer entry und next
}
```

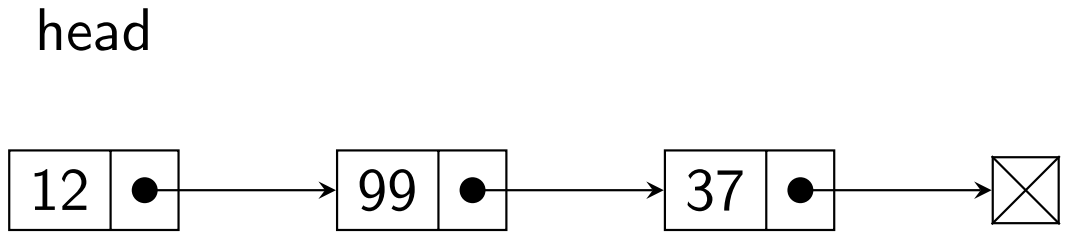
Einfügen eines neuen Knotens

Einfügen eines neuen Knotens mit Eintrag i:

1. Erzeuge neue `ListNode` `newNode` mit `value` i und `nextNode` wird auf `head` gesetzt.
2. Ändere den `head`-Zeiger der Liste auf `newNode`

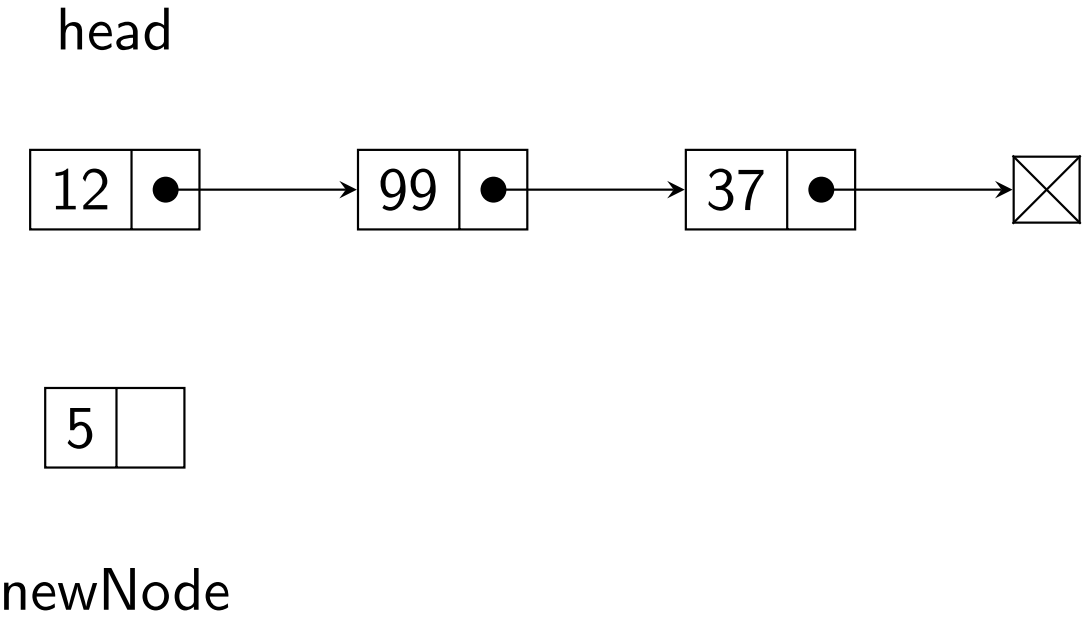
Beispiel Einfügen in verkettete Liste

Wir wollen einen neuen Listen-Eintrag mit dem Wert 5 zu der Beispielsliste hinzufügen:



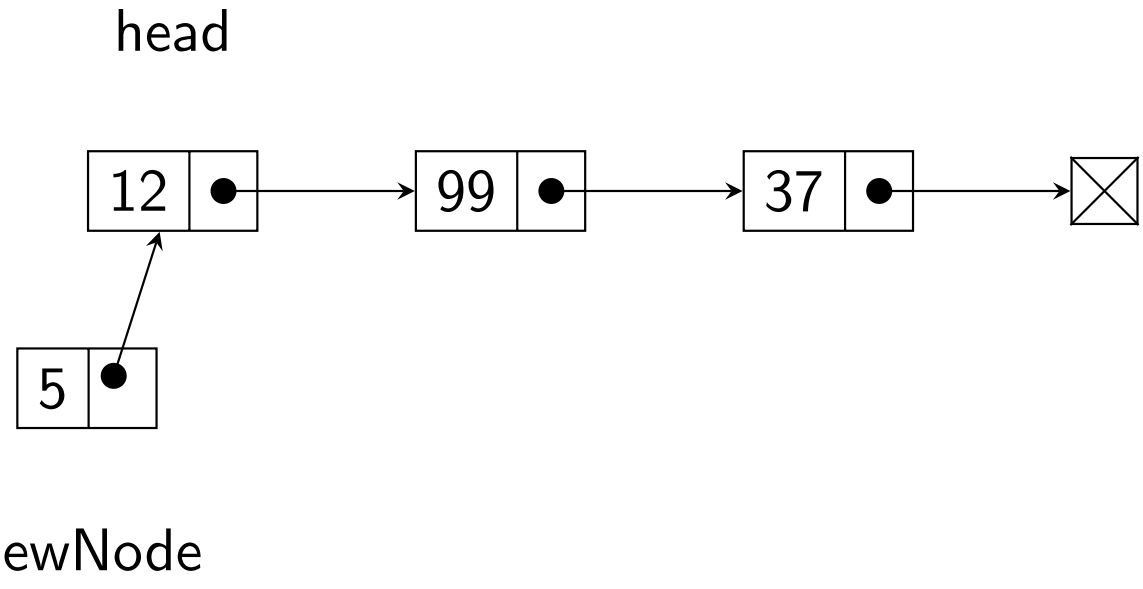
Beispiel Einfügen in verkettete Liste

Dafür erzeugen wir ein neues Listenelement



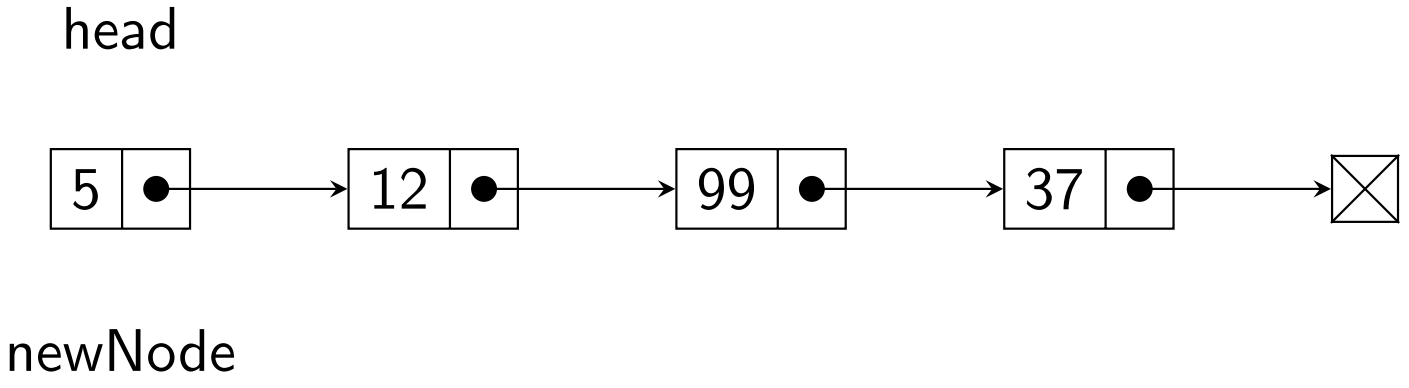
Beispiel Einfügen in verkettete Liste

... dessen `next` Referenz auf den Kopf der Liste zeigt



Beispiel Einfügen in verkettete Liste

Schließlich wird der `head`-Zeiger auf `newNode` umgesetzt.



Implementation Einfügen in verkettete Liste

```
public void add(int value) {
   ListNode newNode = new ListNode(value, this.head);
   this.head = newNode;
}
```

Länge einer verkettete Liste

Die Länge einer Liste ist variabel, daher muss die Länge gezählt werden. Hierzu folgen wir den `next`-Verweisen vom Kopf bis zum Ende der Liste.

```
public int size() {
   ListNode current = this.head;
   int count = 0;
   while (current != null) {
      ++count;
      current = current.getNext();
   }
   return count;
}
```

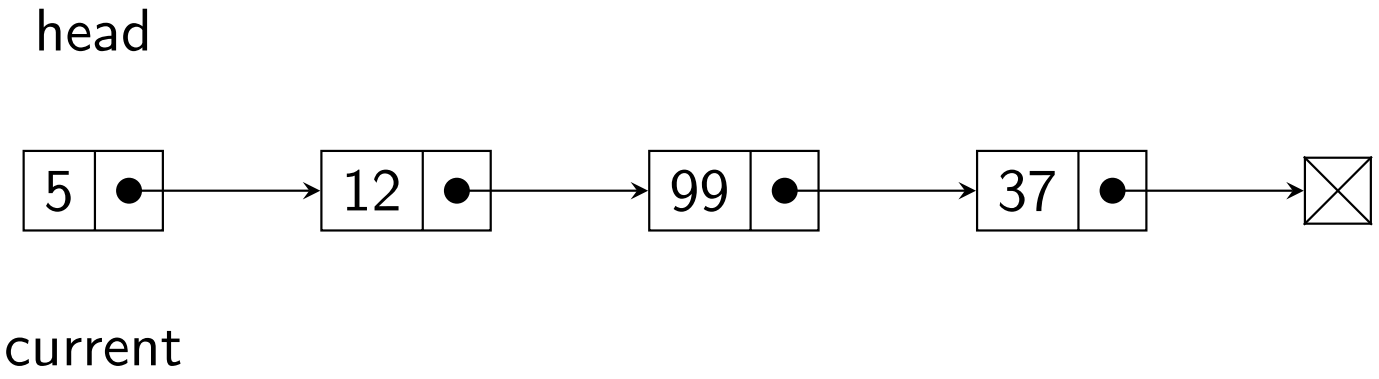
Länge einer verkettete Liste

Analog kann überprüft werden ob eine Liste einen Eintrag `value` enthält

```
public boolean contains(int value) {
   ListNode current = this.head;
   while (current != null) {
      if (current.getEntry() == value)
         return true;
      current = current.getNext();
   }
   return false;
}
```

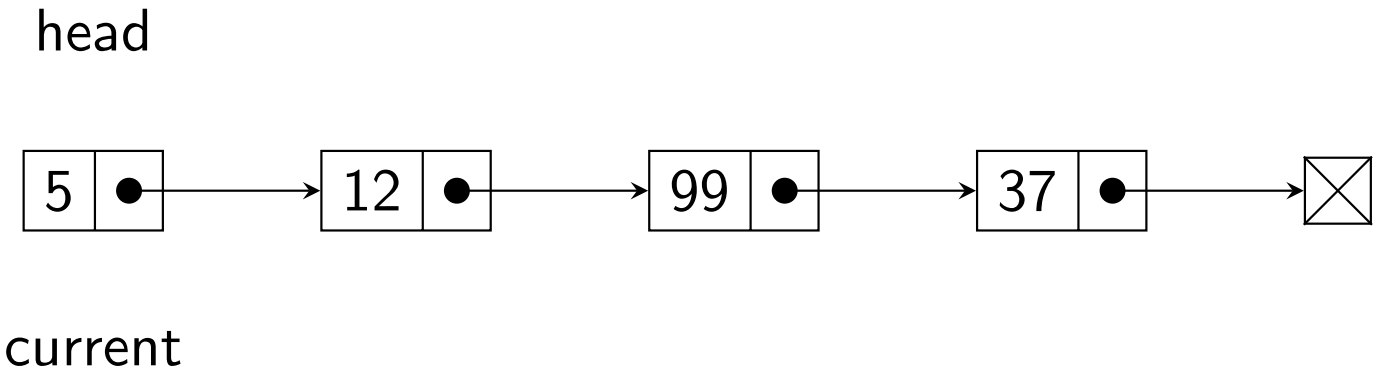
Beispiel Suchen in verkettete Liste

Angenommen wir suchen die 99 in unserer Beispielliste. Wir beginnen beim Kopf der Liste...



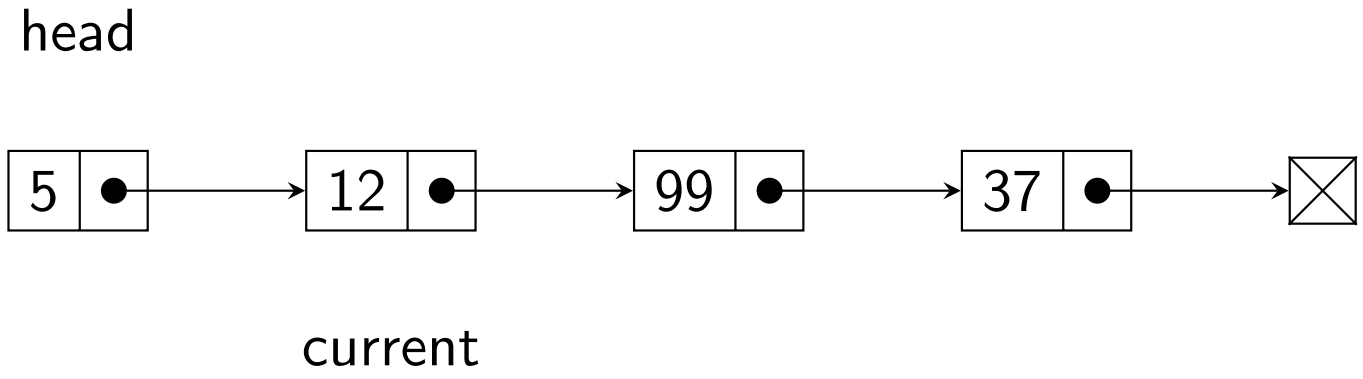
Beispiel Suchen in verkettete Liste

Der Kopf der Liste trägt nicht den Eintrag 99 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



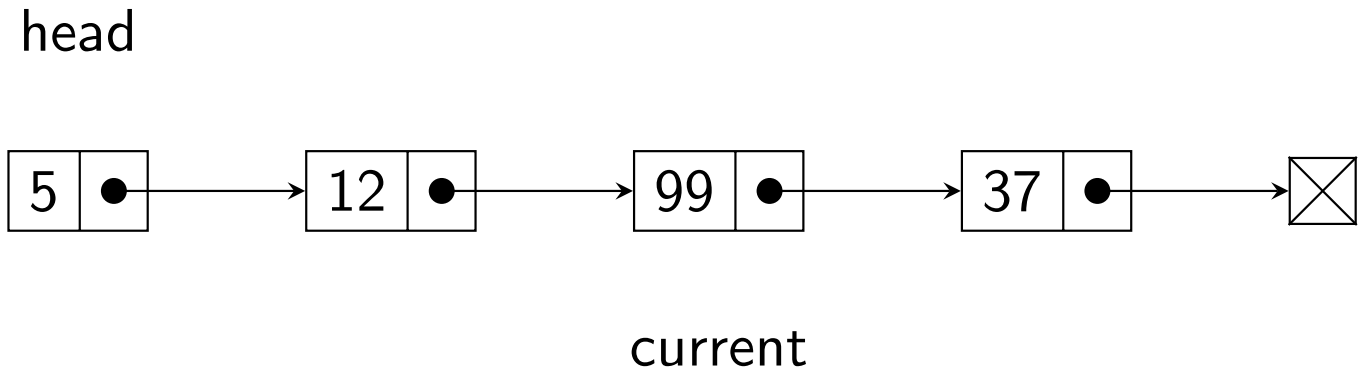
Beispiel Suchen in verkettete Liste

Der zweite Eintrag der Liste trägt nicht den Eintrag 99 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



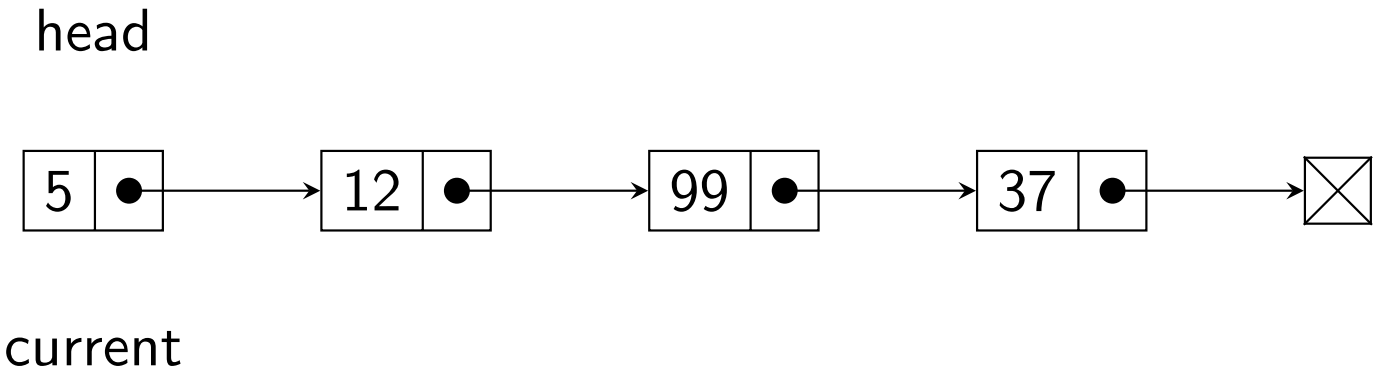
Beispiel Suchen in verkettete Liste

Der dritte Eintrag der Liste trägt den Eintrag 99, wir geben `true` zurück.



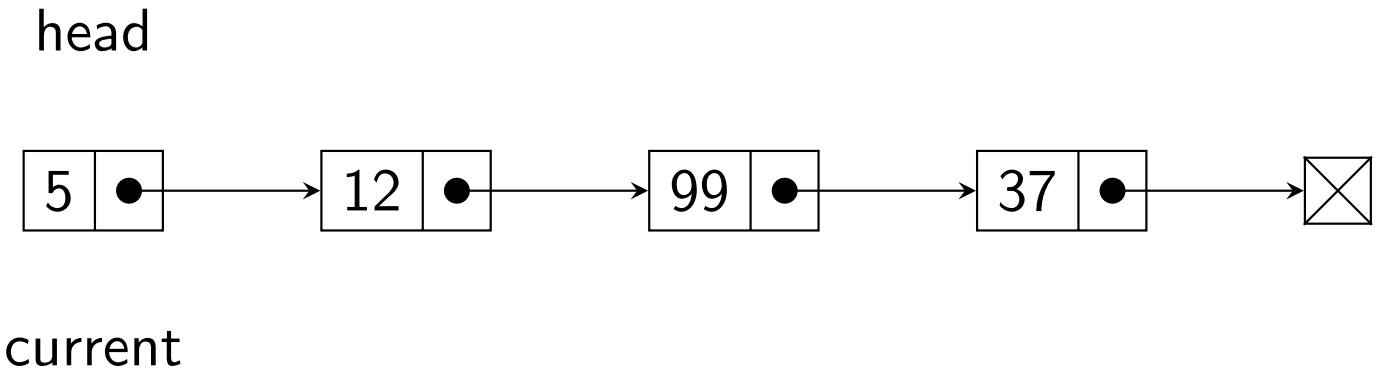
Beispiel Suchen in verkettete Liste

Angenommen wir suchen die 10 in unserer Beispielliste. Wir beginnen beim Kopf der Liste...



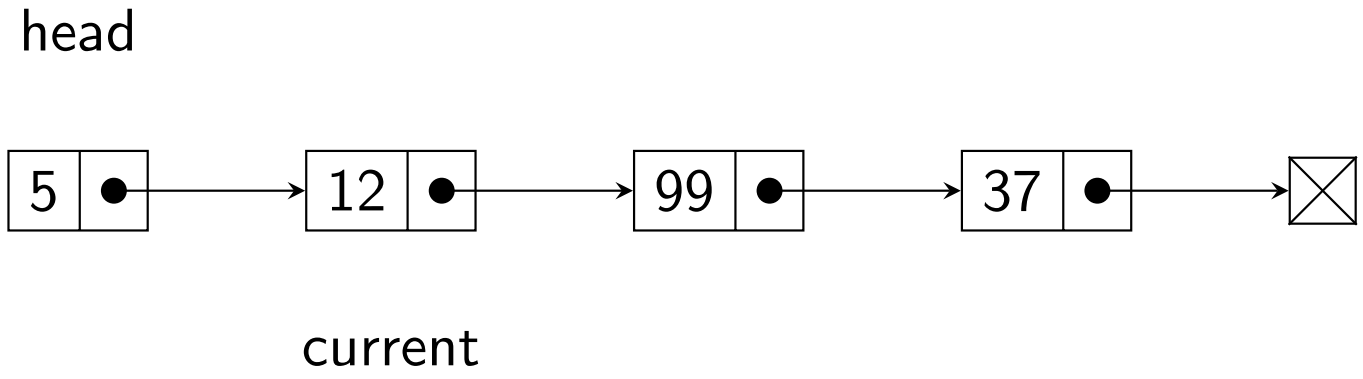
Beispiel Suchen in verkettete Liste

Der Kopf der Liste trägt nicht den Eintrag 10 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



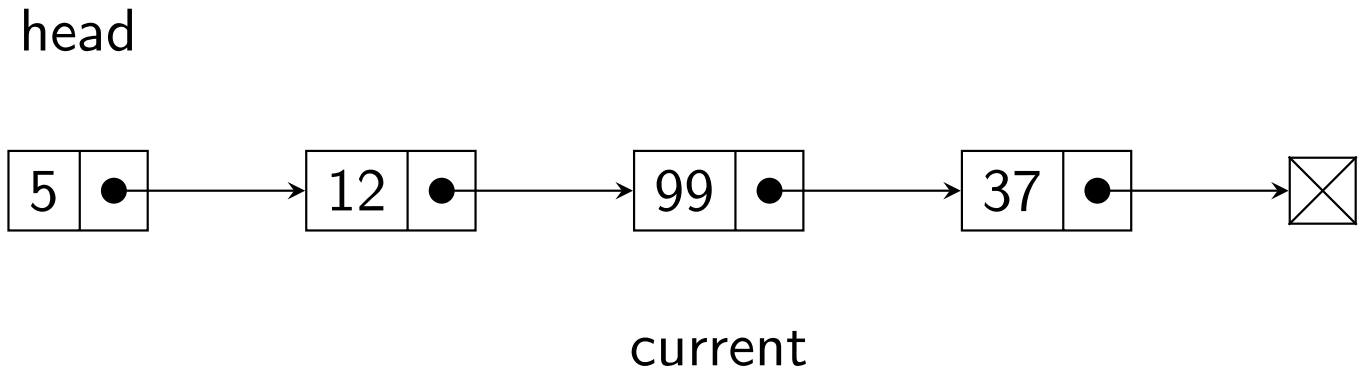
Beispiel Suchen in verkettete Liste

Der zweite Eintrag der Liste trägt nicht den Eintrag 10 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



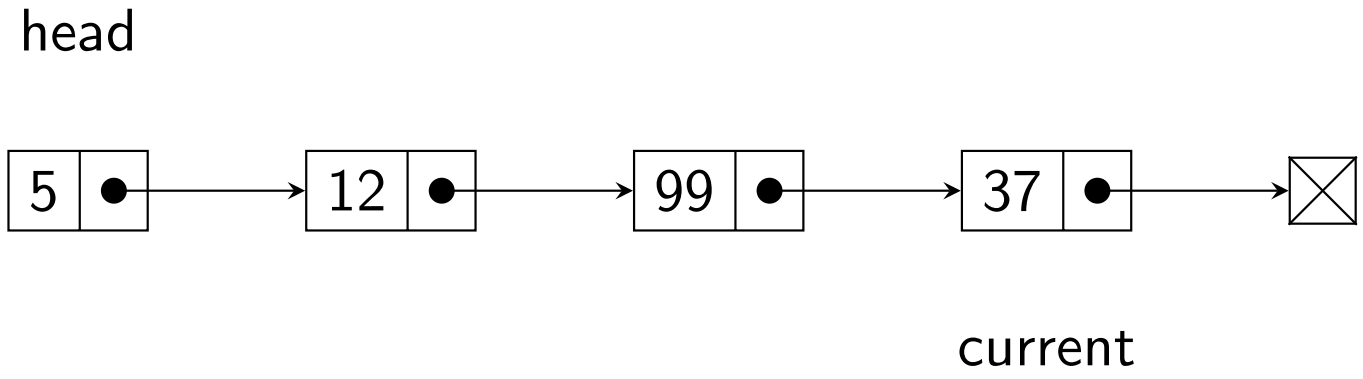
Beispiel Suchen in verkettete Liste

Der dritte Eintrag der Liste trägt nicht den Eintrag 10 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



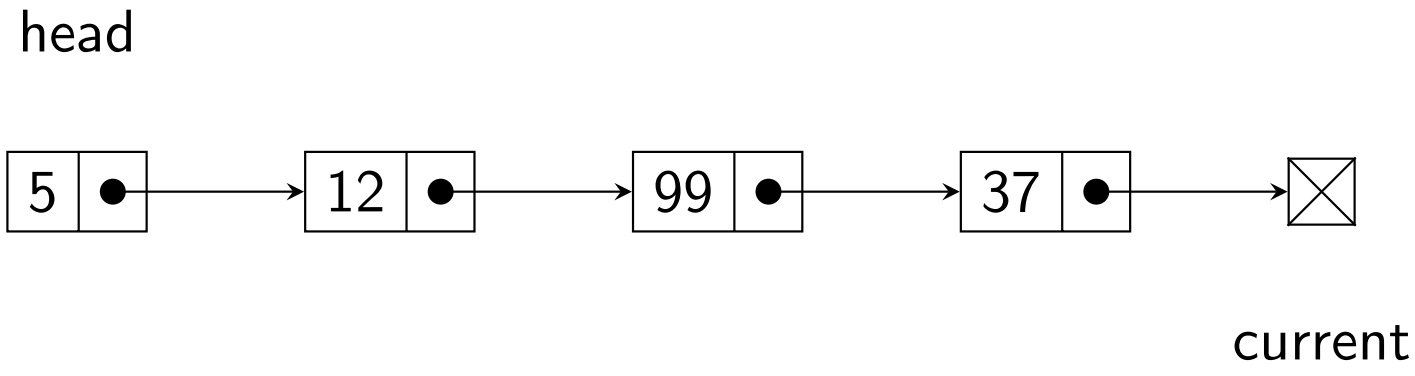
Beispiel Suchen in verkettete Liste

Der vierte Eintrag der Liste trägt nicht den Eintrag 10 und ist nicht der Schwanz der Liste, wir suchen beim Nachfolger weiter.



Beispiel Suchen in verkettete Liste

Der fünfte Eintrag der Liste ist der Schwanz der Liste, wir geben `false` zurück.



Entfernen eines Knotens

Um einen Knoten mit einem bestimmten Eintrag zu entfernen gehen wir wie folgt vor:

1. Suche den entsprechenden Knoten `node` .

Entfernen eines Knotens

Um einen Knoten mit einem bestimmten Eintrag zu entfernen gehen wir wie folgt vor:

1. Suche den entsprechenden Knoten `node` .
2. Falls `node` der Kopf der Liste ist, setze `head` auf `node.next`.
3. Anderenfalls setze den `next`-Verweis seines Vorgängers auf `node.next`.

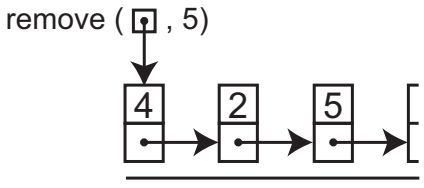
Das lässt sich rekursiv leicht umsetzen.

Löschen aus einer verkettete Liste

```
public void remove(int value) {
   this.head = remove(this.head, value);
}
private ListNode remove(ListNode node, int value) {
   if (node == null)
      return null;
   if (node.getEntry() == value)
      return node.getNext();
   node.setNext(remove(node.getNext(), value));
   return node;
}
```

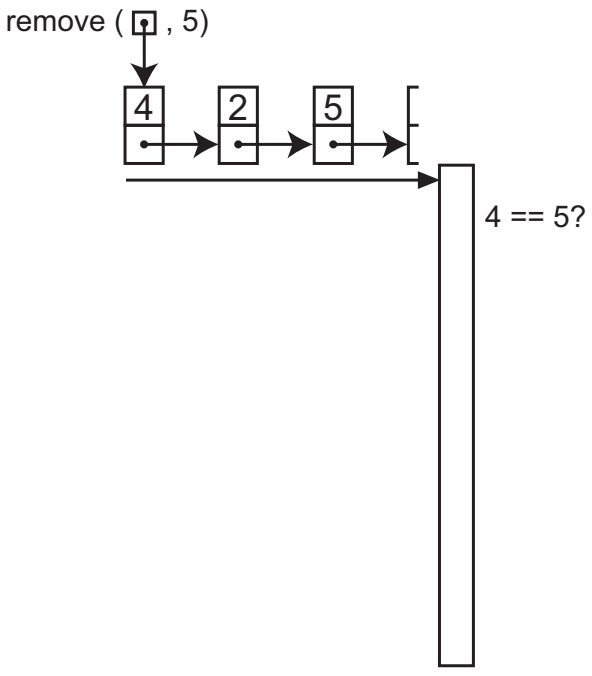
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



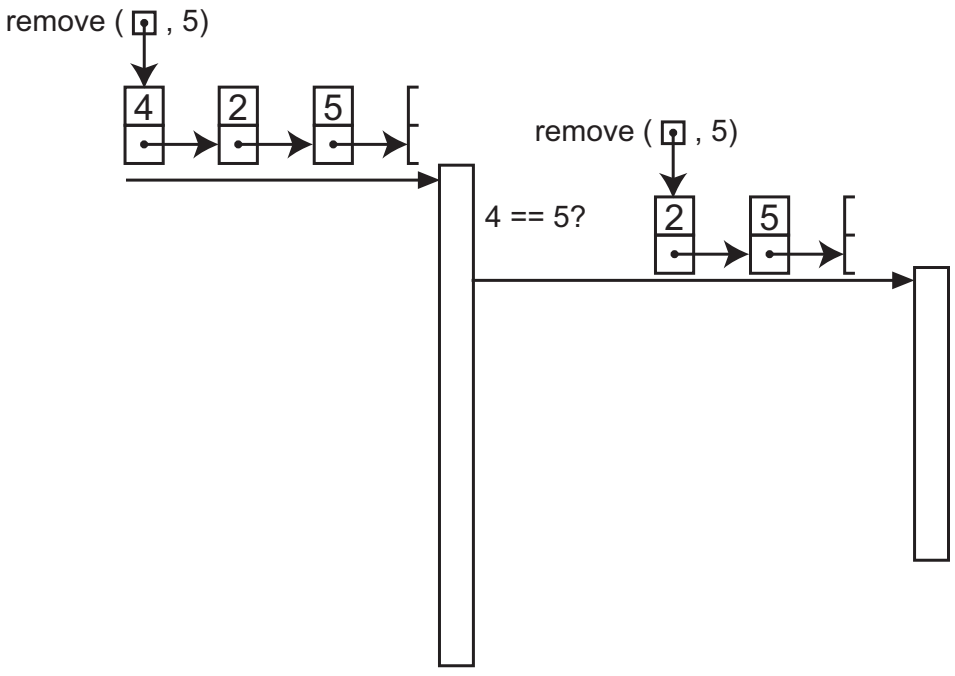
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



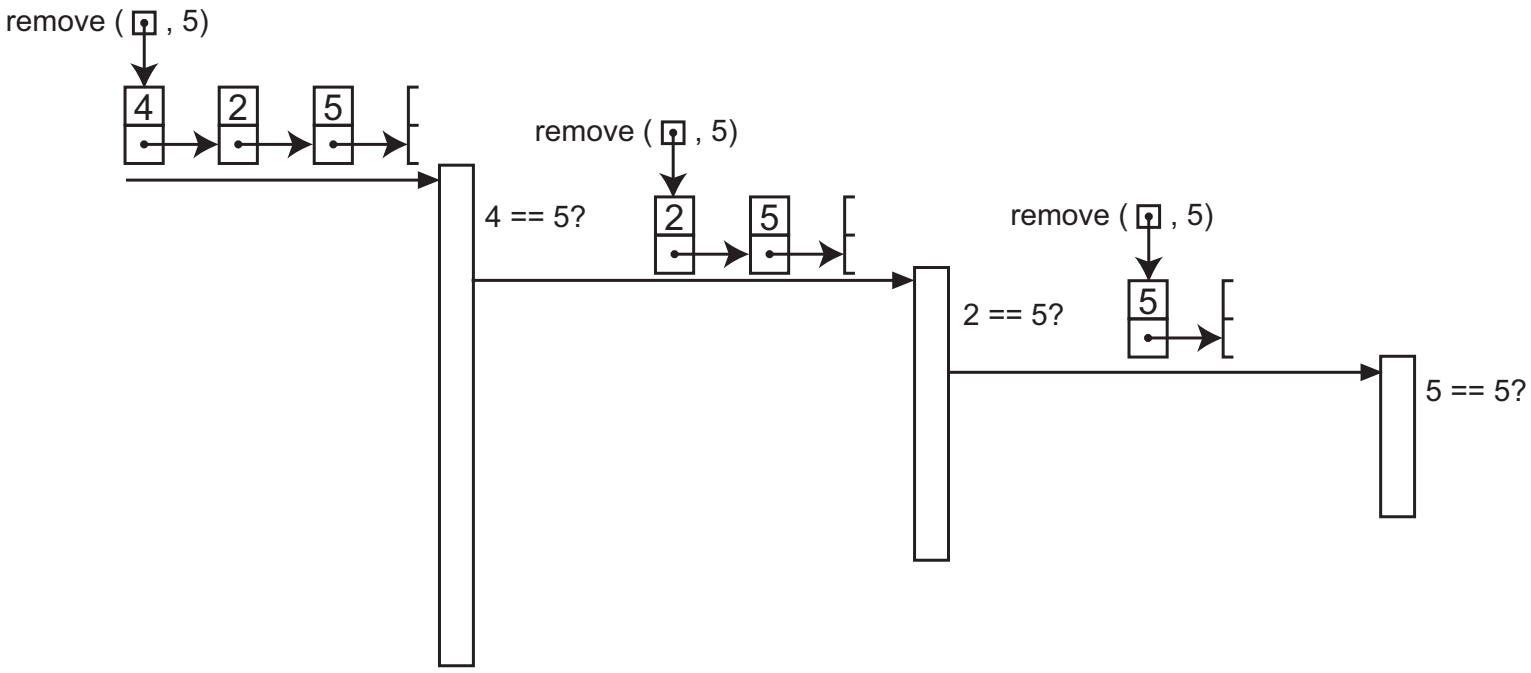
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



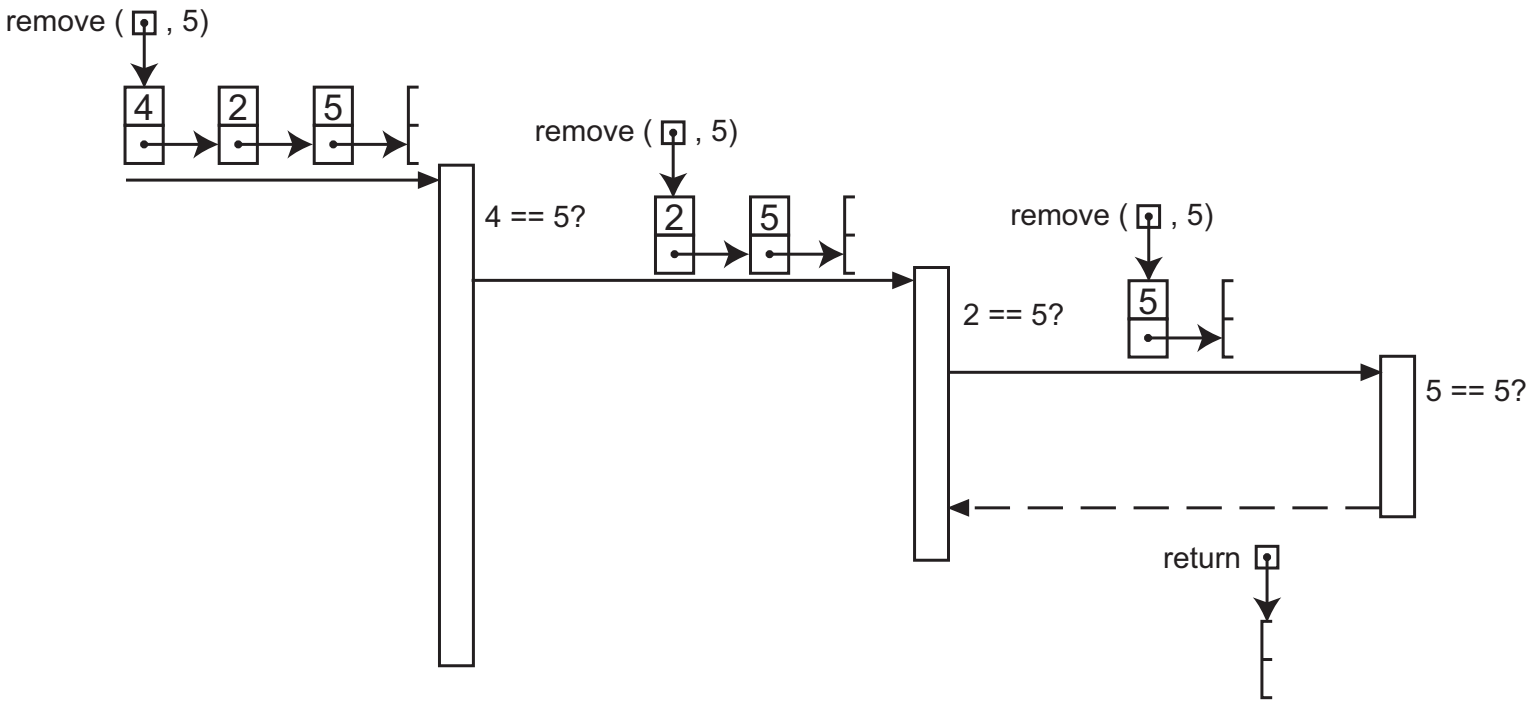
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



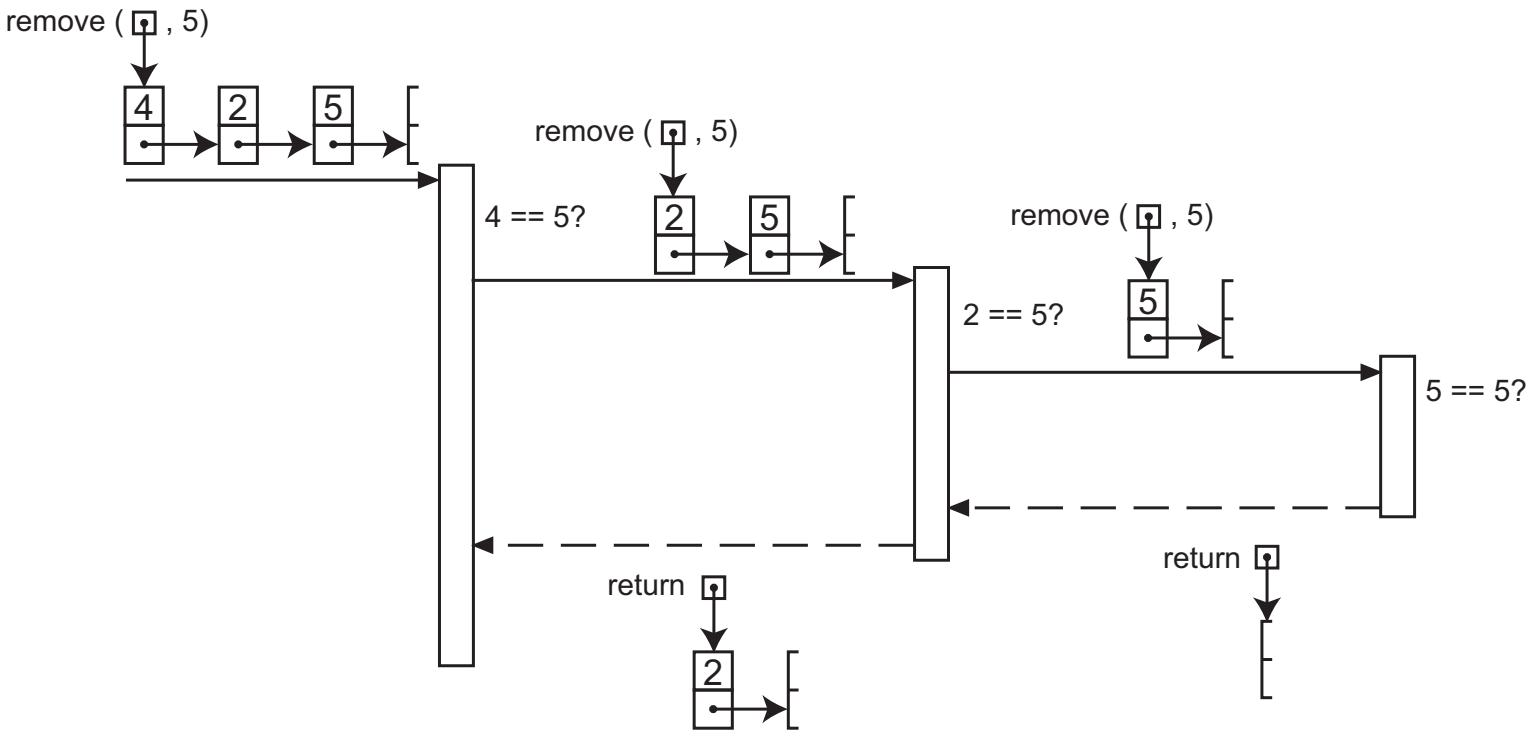
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



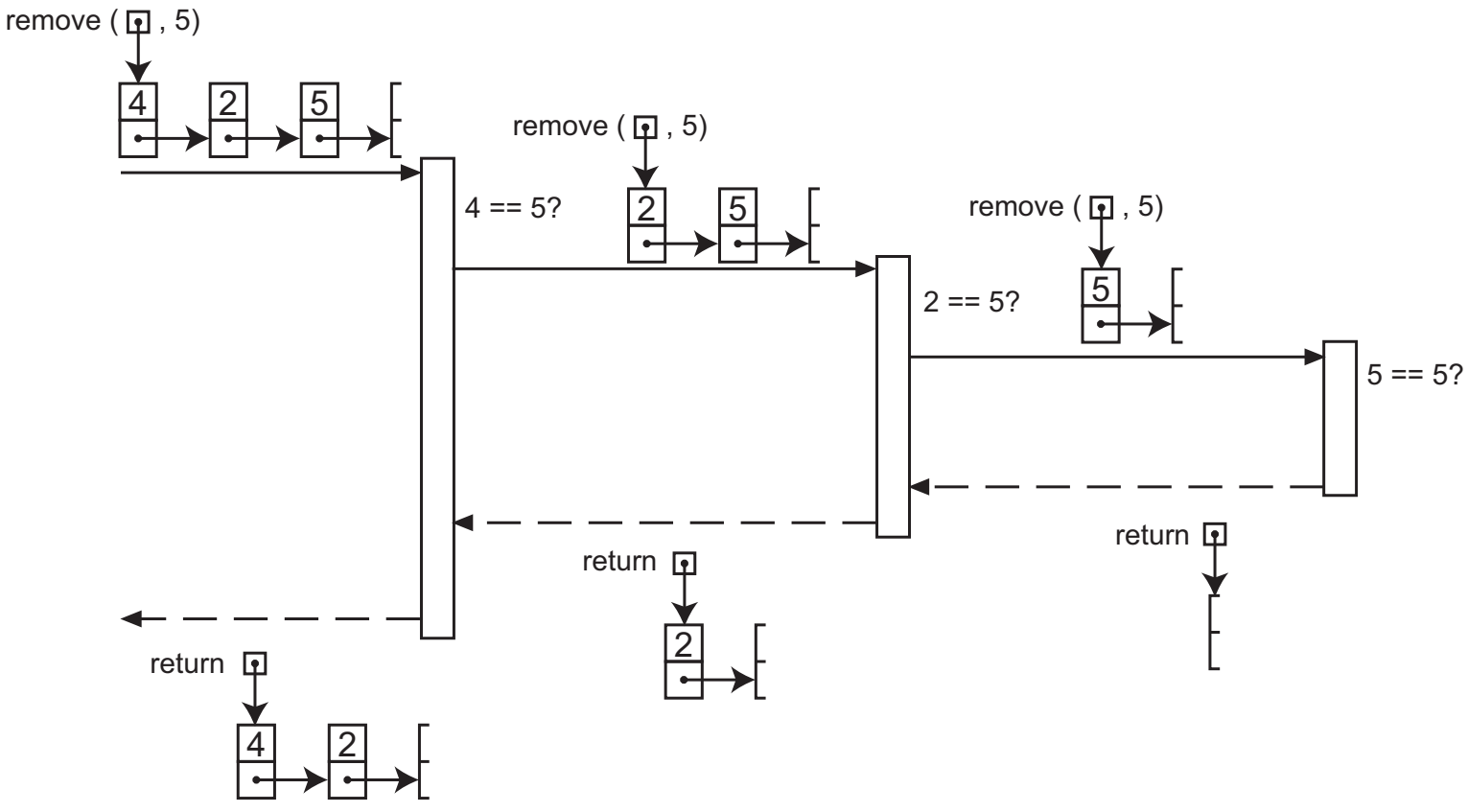
Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



Beispiel: Löschen aus einer Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Löschen



Sortierte Listen

- Oft muss man auf Elemente der Größe nach zugreifen.
- Minimumssuche in einer unsortierten Liste: n Vergleichsschritte
- Minimumssuche in einer sortierten Liste: 0 Vergleichsschritte notwendig, stets der Kopf

Sortierte Listen

- Oft muss man auf Elemente der Größe nach zugreifen.
- Minimumssuche in einer unsortierten Liste: n Vergleichsschritte
- Minimumssuche in einer sortierten Liste: 0 Vergleichsschritte notwendig, stets der Kopf
- Darum: Sortierte Listen
- Schnellerer Zugriff auf Minimum wird durch teureres Einfügen erkauft

Sortierte Listen aufbauend auf Listen

- Implementation der Liste kann wiederverwendet werden (z.B. mittels Vererbung! Beachte Implementierungsunterschied zum Skript)
- `add` muss neu implementiert werden
- `contains` und `remove` können optimiert werden: Sobald ein größeres Element als das gesuchte in der Liste auftaucht, kann `false` zurückgegeben werden.

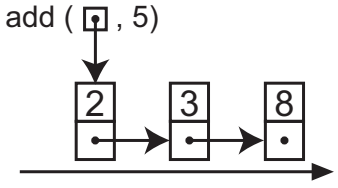
Einfügen in sortierte Liste

Suche das erste Element das größer ist als das einzufügende und füge das einzufügende Element davor ein.

```
public void add(int value) {
   this.head = add(this.head, value);
}
private ListNode add(ListNode node, int value) {
   if (node == null)
      return new ListNode(value, node);
   if (node.getEntry() > value)
      return new ListNode(value, node);
   node.setNext(add(node.getNext(), value));
   return node;
}
```

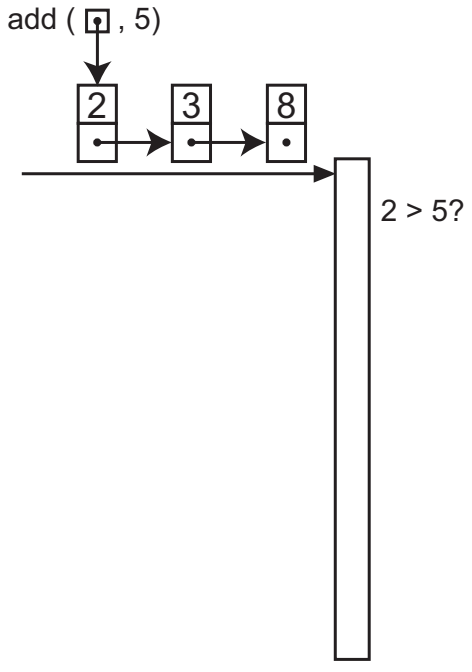
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



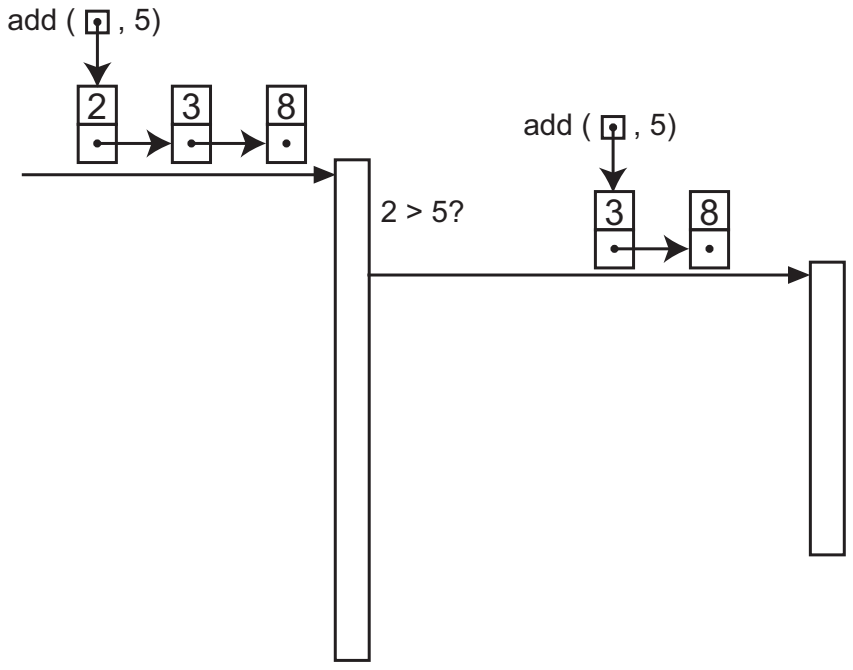
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



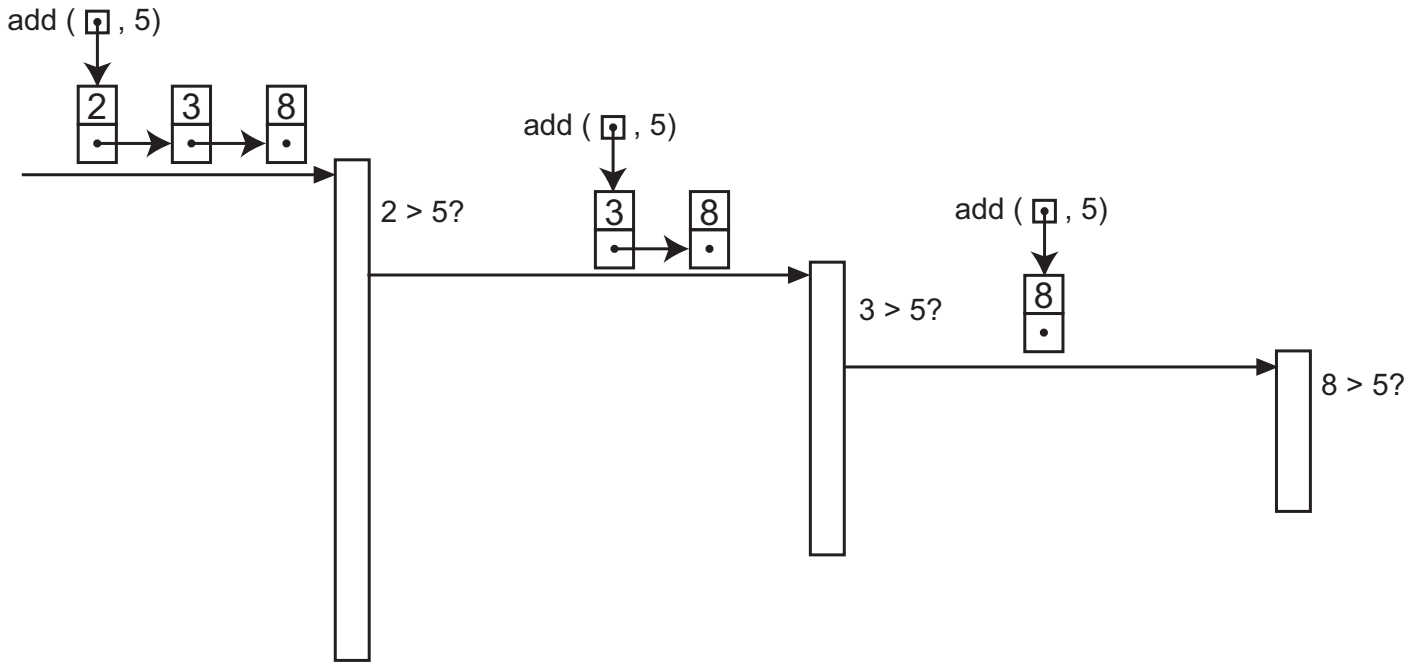
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



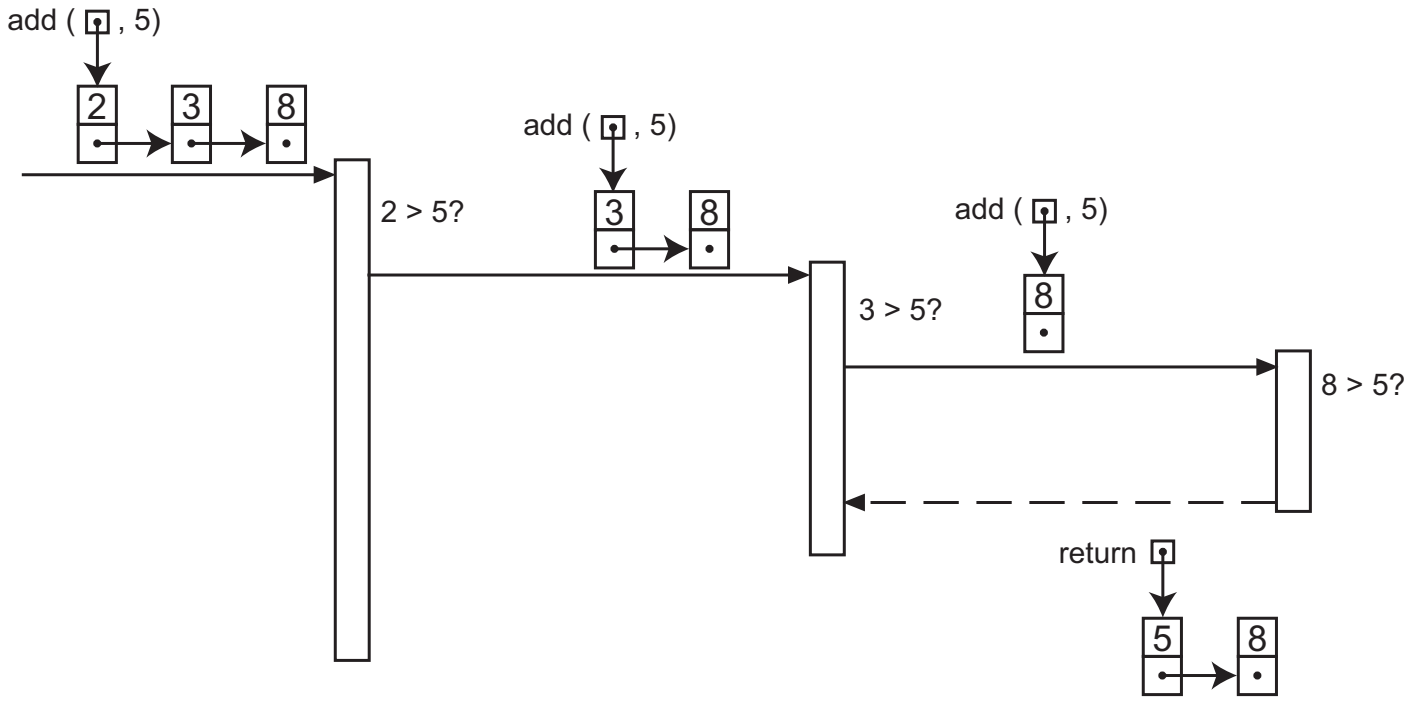
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



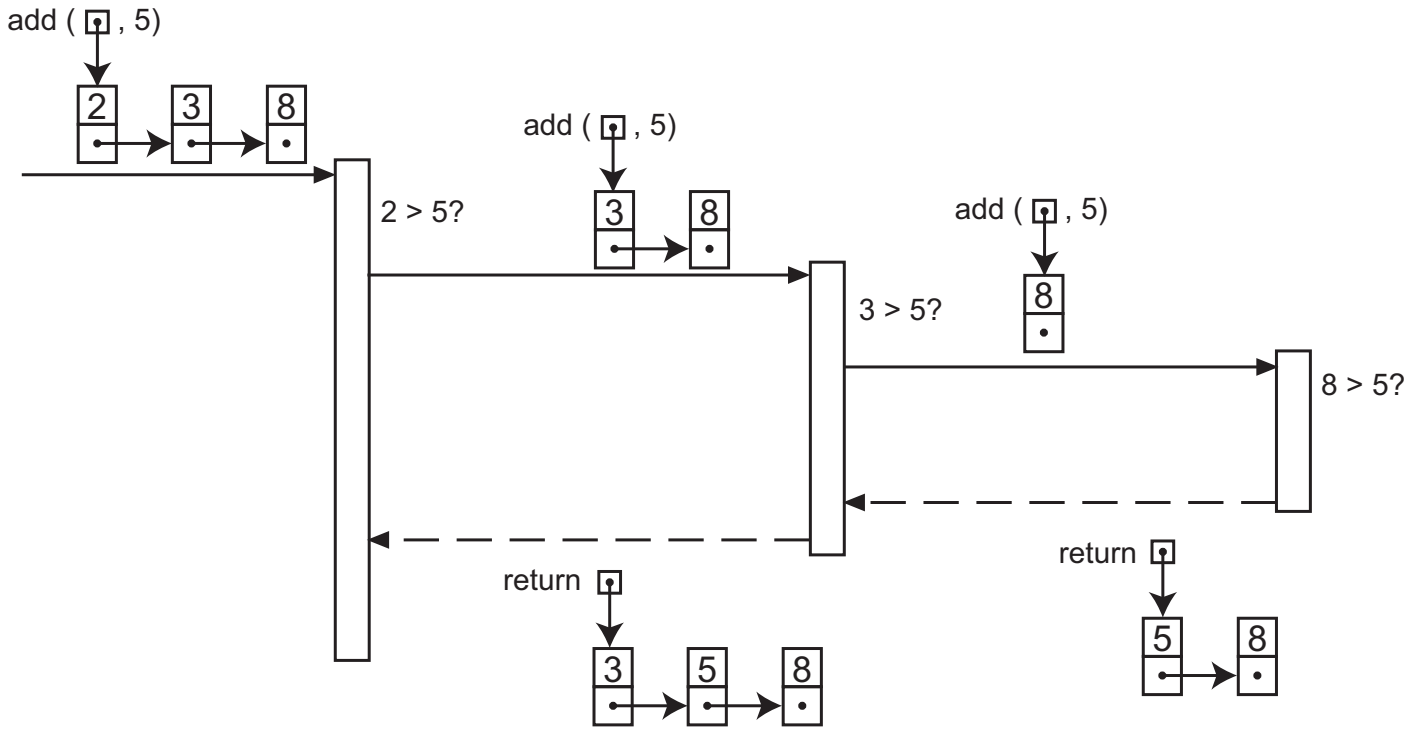
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



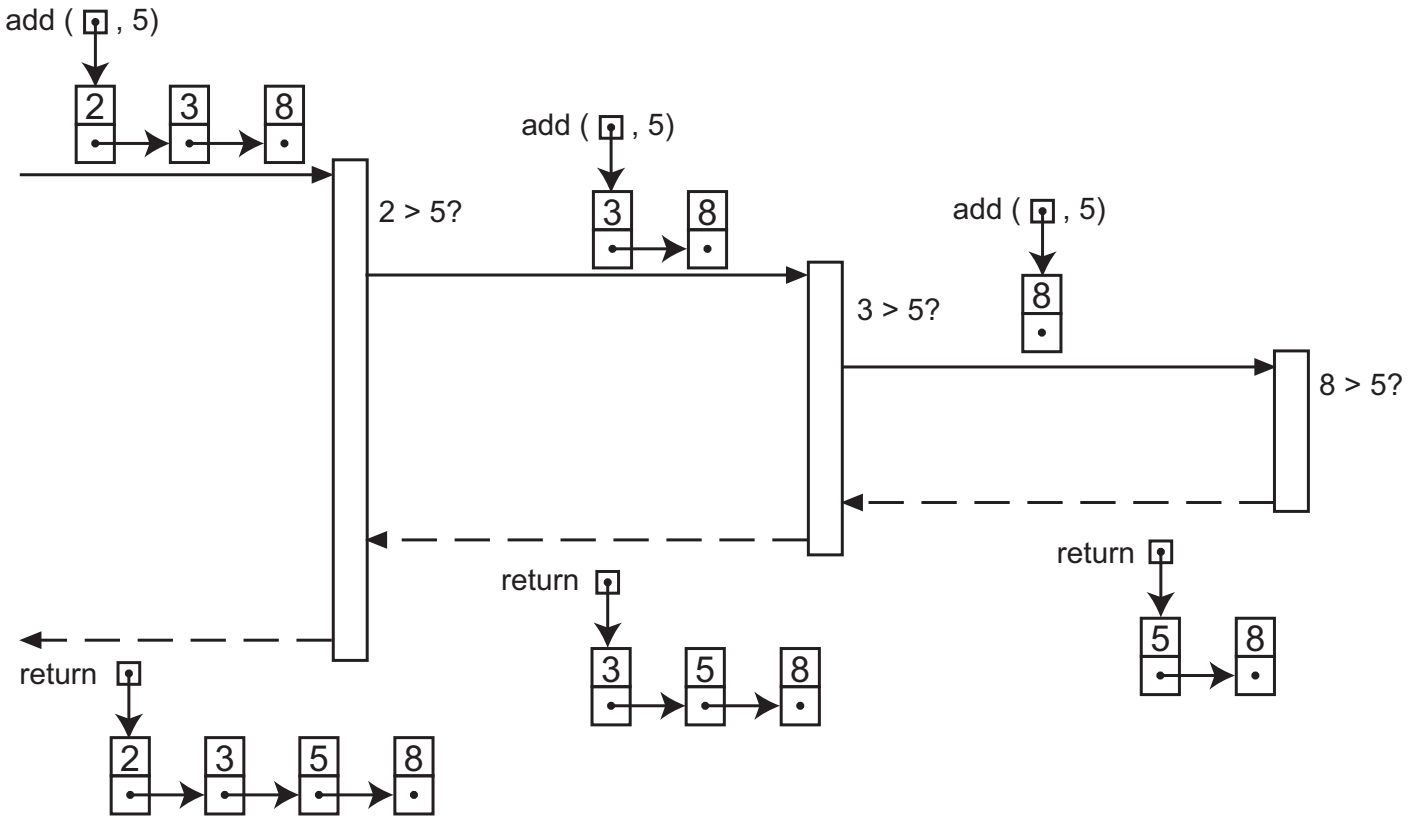
Beispiel: Einfügen in sortierte Liste

Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



Beispiel: Einfügen in sortierte Liste

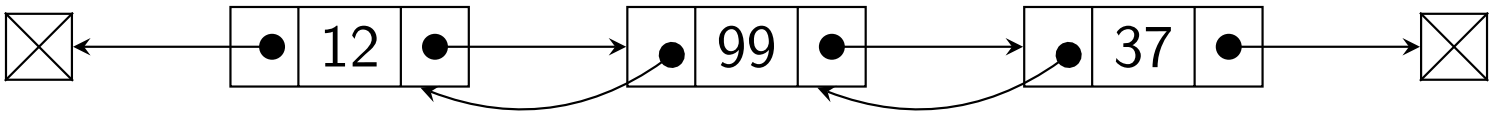
Die folgende Grafik zeigt den rekursiven Ablauf beim Einfügen



Doppelt verkettete Liste

- Einfach verkettete Listen sind teuer bei Operationen hinten in der Liste
- Beispiel: Möchte man hinten ein Element anfügen, muss man erst zum Schwanz laufen
- Mögliche Lösung: Eigene Variable für den Schwanz
- Verschiebt das Problem aber nur: Beim Löschen hinten in der Liste braucht man den Vorgänger des Schwanzes
- Daher: Doppelt verkettete Liste
- Funktioniert wie einfach verkettete Liste, jeder Eintrag speichert aber Vorgänger und Nachfolger

Beispiel doppelt verkettete Liste



Weitere lineare Datenstrukturen: Stapel (engl. Stack)

- Man stelle sich einen (z.B. Bücher-)Stapel vor

Weitere lineare Datenstrukturen: Stapel (engl. Stack)

- Man stelle sich einen (z.B. Bücher-)Stapel vor
- Neue Elemente werden stets oben (d.h. vorne) angefügt
- Vom Stapel entnehmen kann man stets nur von oben (d.h. vorne)
- Auch lesend kann stets nur auf das oberste Element zugegriffen werden

Weitere lineare Datenstrukturen: Stapel (engl. Stack)

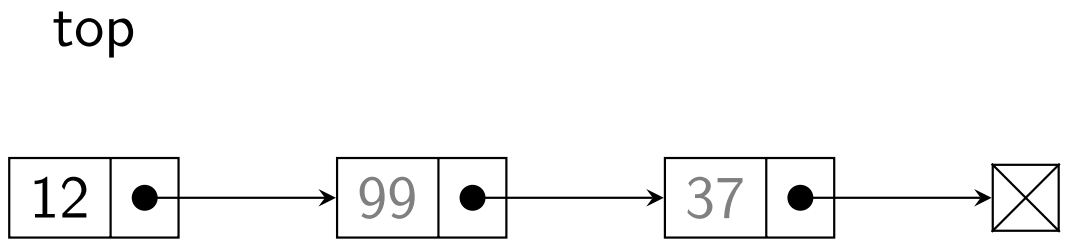
- Man stelle sich einen (z.B. Bücher-)Stapel vor
- Neue Elemente werden stets oben (d.h. vorne) angefügt
- Vom Stapel entnehmen kann man stets nur von oben (d.h. vorne)
- Auch lesend kann stets nur auf das oberste Element zugegriffen werden
- Wir nennen das das LIFO-Prinzip (engl. last in first out)
- Implementierung effizient möglich als Containerklasse einer einfach verketteten Liste

Weitere lineare Datenstrukturen: Stapel (engl. Stack)

- Man stelle sich einen (z.B. Bücher-)Stapel vor
- Neue Elemente werden stets oben (d.h. vorne) angefügt
- Vom Stapel entnehmen kann man stets nur von oben (d.h. vorne)
- Auch lesend kann stets nur auf das oberste Element zugegriffen werden
- Wir nennen das das LIFO-Prinzip (engl. last in first out)
- Implementierung effizient möglich als Containerklasse einer einfach verketteten Liste
- Anwendung beispielsweise in Hardware-naher Programmierung (Methodenstapel) und Theoretischer Informatik (Kellerautomat)

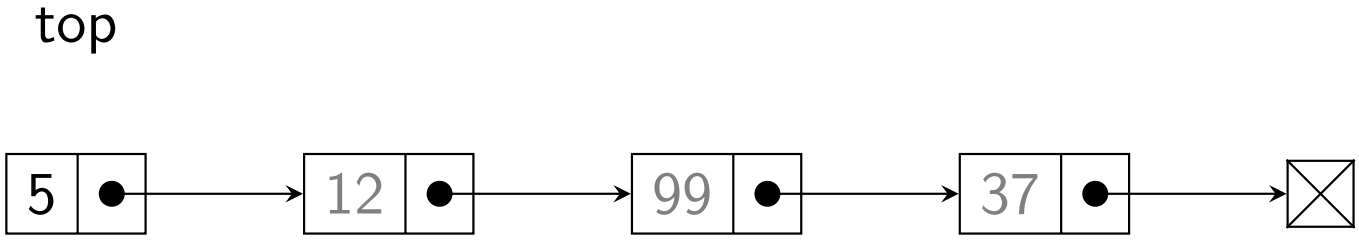
Beispiel Einfügen in Stapel

Wir wollen einen neuen Eintrag mit dem Wert 5 zum Stapel (Operation `push`) hinzufügen:



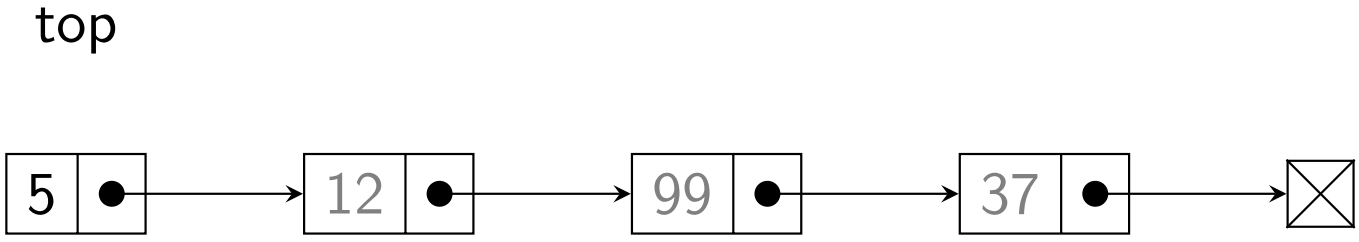
Beispiel Einfügen in Stapel

... anschließend ist das alte `top`-Element nicht mehr zugreifbar, sondern nur noch das neue `top`-Element.



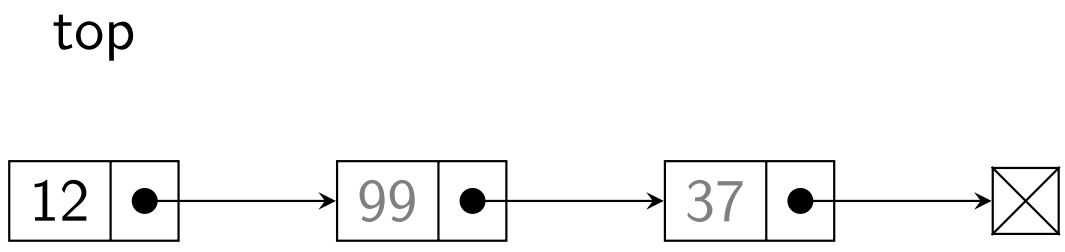
Beispiel Entfernen aus Stapel

Umgekehrt, wenn wir nun vom Stapel entfernen wollen, können wir das mit der Operation pop tun:



Beispiel Entfernen aus Stapel

... anschließend ist das vormals zweite Elemente zugreifbar.



Weitere lineare Datenstrukturen: Warteschlange (engl. Queue)

- Man stelle sich eine Warteschlange vor

Weitere lineare Datenstrukturen: Warteschlange (engl. Queue)

- Man stelle sich eine Warteschlange vor
- Neue Elemente werden stets hinten angefügt
- Elemente verlassen die Schlange stets von vorne
- Auch lesend kann stets nur auf das vorderste Element zugegriffen werden

Weitere lineare Datenstrukturen: Warteschlange (engl. Queue)

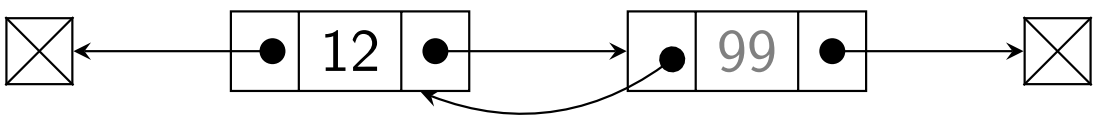
- Man stelle sich eine Warteschlange vor
- Neue Elemente werden stets hinten angefügt
- Elemente verlassen die Schlange stets von vorne
- Auch lesend kann stets nur auf das vorderste Element zugegriffen werden
- Wir nennen das das FIFO-Prinzip (engl. first in first out)
- Implementierung effizient möglich als Containerklasse einer doppelt verketteten Liste

Weitere lineare Datenstrukturen: Warteschlange (engl. Queue)

- Man stelle sich eine Warteschlange vor
- Neue Elemente werden stets hinten angefügt
- Elemente verlassen die Schlange stets von vorne
- Auch lesend kann stets nur auf das vorderste Element zugegriffen werden
- Wir nennen das das FIFO-Prinzip (engl. first in first out)
- Implementierung effizient möglich als Containerklasse einer doppelt verketteten Liste
- Anwendung beispielsweise beim Scheduling (Betriebssysteme)

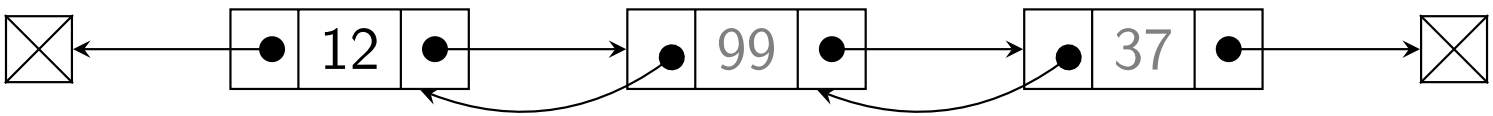
Anfügen an Warteschlange

Wir wollen einen Eintrag mit dem Wert 37 an die Warteschlange anfügen (`enqueue`):



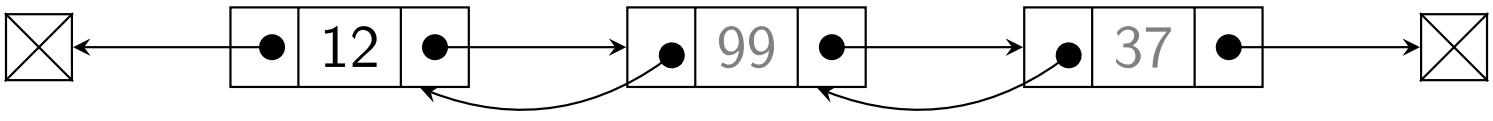
Anfügen an Warteschlange

Man beachte, dass unverändert der erste Eintrag mit der 12 der einzig zugreifbare ist:



Entfernen aus Warteschlange

Wollen wir umgekehrt ein Element aus der Schlange entfernen (`dequeue`), muss dieser Zugriff vorne geschehen:



Entfernen aus Warteschlange

... anschließend ist das vormals zweite Element der Schlange zugreifbar:

Vor- und Nachteile der dynamischen Datenstrukturen

+ Flexibler einsetzbar als Felder
+ Einfacher zu warten als Felder
– Weniger effizient als Felder

Fragen zur Vorlesungseinheit

1. Wie fügt man Elemente zu einer einfach verketteten Liste hinzu und entfernt sie?
2. Welchen Vorteil haben die hier vorgestellten Datenstrukturen im Vergleich zu einem Feld?
3. Was bedeuten LIFO und FIFO?