Kapitel 2: Datentyp Liste

- Einleitung
- Listen-Interface
- Liste als Feld: ArrayList
- Einfach verkettete Listen
- Hilfskopfknotentechnik
- Liste als einfach verkettete Liste: LinkedList
- Doppelt verkettete Listen

Listen und ihre Operationen

Definition

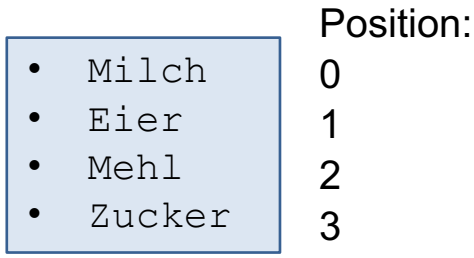
- Eine Liste der Länge n ist eine endliche Folge von n Elementen:

  $a_0, a_1, \ldots, a_{n-1}$

- Die Elemente einer Liste der Länge n besitzen eine Position: 0, 1, ..., n-1.

Beispiele:

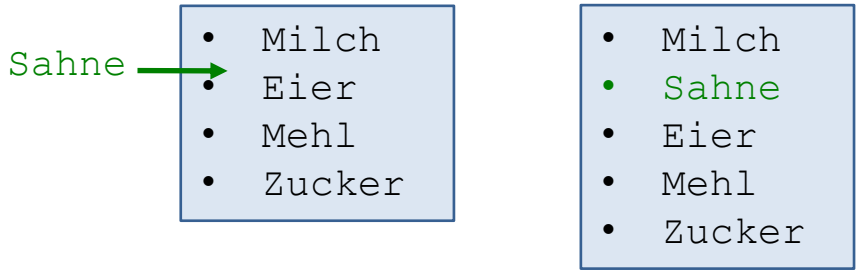
- Einkaufslisten
- Tagesordnungen
- ...



Typische Operationen:

- einfügen an eine Position
- löschen einer Position
- lesen einer Position
- ändern einer Position

Einfügen von Sahne an Position 1:



Listen-Interface

```
public interface List {
    void add(int x);
    void add(int idx, int x);
    int set(int idx, int x);
    int get(int idx);
    void remove(int idx);
    int size();
    void clear();
    boolean isEmpty();
}
```

- list.add(x) fügt x in die Liste list hinten an; entspricht list.add(list.size(), x)
- list.add(i, x) fügt an Position i das Element x ein.
- list.set(i, x) überschreibt das Element an der Position i mit x.
  Der alte Wert wird zurückgeliefert.
- list.get(i) liefert den Wert an der Position i.
- list.remove(i) löscht das Element an der Position i.
- list.size() ist die Anzahl Elemente in der Liste list.
- list.clear() löscht alle Elemente.
- list.isEmpty() prüft, ob die Liste list leer ist.

Jetzt nur int-Liste; generische Liste später

- In diesem Kapitel sollen vorerst nur int-Listen behandelt werden.
- Die Erweiterung auf beliebige Element-Typen geschieht später als generischer Listentyp.

Generisches Listen-Interface:

```
public interface List<E> {
    void add(E x);
    void add(int idx, E x);
    int set(int idx, E x);
    E get(int idx);
    void remove(int idx);
    int size();
    void clear();
    boolean isEmpty();
}
```

E steht für einen beliebigen Elementyp

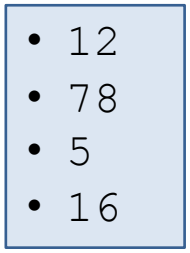
Kapitel 2: Datentyp Liste

- Einleitung
- Listen-Interface
- Liste als Feld: ArrayList
- Einfach verkettete Listen
- Hilfskopfknotentechnik
- Liste als einfach verkettete Liste: LinkedList
- Doppelt verkettete Listen

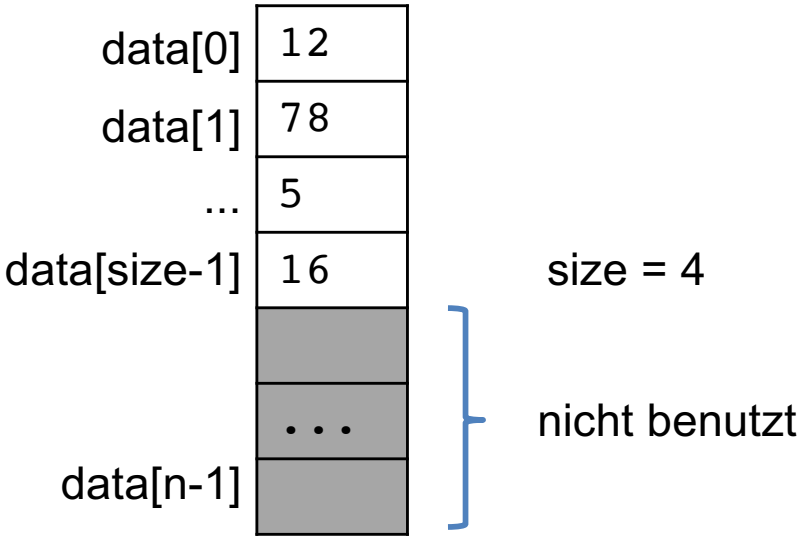
Liste als Feld: class ArrayList (1)

- Die Realisierung einer Liste als Feld ist naheliegend
- Elemente werden lückenlos im Feld gehalten.
  Beim Einfügen bzw. Löschen müssen die Elemente verschoben werden.
- Falls das Feld gefüllt ist, muss das Feld vergrößert werden,
  d.h. umkopieren in ein größeres Feld.

Liste



Liste als Feld:



Liste als Feld: class ArrayList (2)

```
public class ArrayList implements List {

    public ArrayList() {
        clear();
    }

    public final void clear() {
        size = 0;
        data = new int[DEF_CAPACITY];
    }

    public int size() {return size;}

    public boolean isEmpty() {return size == 0;}

    // ...

    private static final int DEF_CAPACITY = 32;
    private int size;
    private int[] data;
}
```

Methode clear ist final und kann damit durch Klassenerweiterungen nicht mehr überschrieben werden.

Wichtig, da das Verhalten des Konstruktors sich nicht mehr ändern darf.

Liste als Feld: class ArrayList (3)

```
public class ArrayList implements List {
    // ...

    public void add(int x) {add(size(), x);}

    public void add(int idx, int x) {
        if (idx < 0 || idx > size)
            throw new IndexOutOfBoundsException();
        if (data.length == size)
            data = Arrays.copyOf(data, 2*size);
        for (int i = size; i > idx; i--) {
            data[i] = data[i-1];
        }
        data[idx] = x;
        size++;
    }

    // ...
}
```

Argument prüfen.

Feld verdoppeln, falls kein Platz mehr ist.

Elemente um eine Position nach rechts verschieben.

Liste als Feld: class ArrayList (4)

```
public class ArrayList implements List {

    // ...
    public int set(int idx, int x) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        int old = data[idx];
        data[idx] = x;
        return old;
    }

    public int get(int idx) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        return data[idx];
    }

    public void remove(int idx) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        for (int i = idx; i < size-1; i++)
            data[i] = data[i+1];
        size--;
    }
}
```

Elemente um eine Position nach links verschieben.

Liste als Feld: class ArrayList (5)

```
public class ArrayList implements List {

    // ...

    @Override
    public String toString() {
        StringBuilder s = new StringBuilder("");
        for (int i = 0; i < size; i++) {
            s.append(data[i]).append(", ");
        }
        s.append("size = ").append(size);
        return s.toString();
    }
}
```

Liste als Feld: class ArrayList (6)

```
public static void main(String[] args) {
      List l = new ArrayList();
      l.add(5);
      l.add(3);
      l.add(2);
      l.add(0,12);
      System.out.println(l);

      System.out.println(l.get(1));
      System.out.println(l.set(1,13));
      System.out.println(l.get(1));
      System.out.println(l);

      l.remove(1);
      System.out.println(l);
 }
```

```
12, 5, 3, 2, size = 4
```

```
5
5
13
12, 13, 3, 2, size = 4
```

```
12, 3, 2, size = 3
```

Aufgaben zu ArrayList

Aufgabe 2.1

Erweitern Sie das Interface `List` und die Klasse `ArrayList` um eine Methode `append(List list)`, die eine weitere Liste list anhängt.

Aufgabe 2.2

Realisieren Sie einen Konstruktor `ArrayList(List list)`, der die Liste mit list initialisiert.

Bemerkungen

- Dynamisch wachsende Listen erfordert teures Kopieren
  in größere Felder.
  Falls Feldgrößen verdoppelt werden (wie hier), ist das Umkopieren nicht so häufig notwendig.
  Frage: wie oft muss das Feld verdoppelt, werden, damit ca. 1 Million Elemente abgespeichert werden können, wenn die Anfangsgröße $2^7 = 128$ ist?
  Die Verdopplung der Feldgrößen geht jedoch auf Kosten von nicht benötigtem Speicherplatz.

- Bei schrumpfenden Listen findet keine Anpassung der Felder statt; das ließe sich jedoch durch Umkopieren in kleinere Felder realisieren.

- Zugriff und Ändern von Elementen ist sehr effizient gelöst.

- Einfügen und Löschen von Elementen ist mit teurem Verschieben verbunden. Besonders teuer ist das Einfügen und Löschen am Listenanfang.

- Daher sind oft verkettete Listen (nächste Folie) die bessere Alternative.

Kapitel 2: Datentyp Liste

- Einleitung
- Listen-Interface
- Liste als Feld: ArrayList
- Einfach verkettete Listen
- Hilfskopfknotentechnik
- Liste als einfach verkettete Liste: LinkedList
- Doppelt verkettete Listen

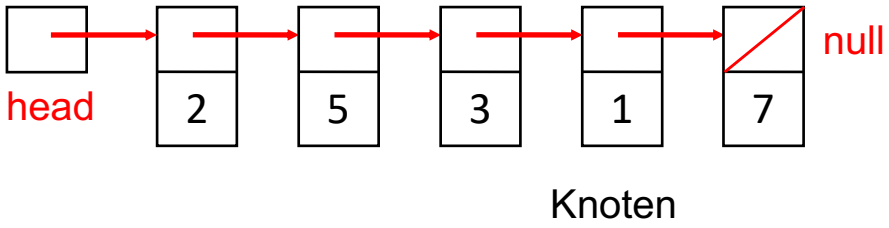
Verkettete Listen

Begriffe

- Eine linear verkettete Liste besteht aus einer Folge von Elementen, die über jeweils eine Referenz auf das nächste Element verkettet sind.
- Element und Referenz auf das nächste Element nennt man auch Knoten (engl. node).
- Es gibt eine Referenz auf den ersten Knoten: wird oft head genannt.
- Letzter Knoten enthält die null-Referenz.

Beispiel

- Liste mit 5 Knoten:



Knoten

- leere Liste:

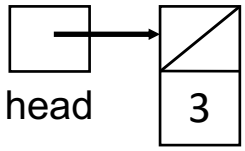
Datentyp Node

Knotentyp Node:

```
class Node {

    Node next;
    int data;

    Node(int x, Node p) {
        data = x;
        next = p;
    }
}
```

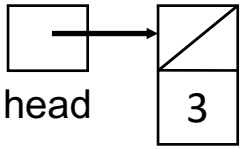
Knoten erzeugen:

```
Node head = new Node(3,null);
```

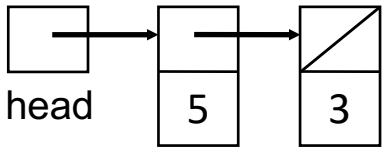


Liste aufbauen durch Einfügen am Anfang

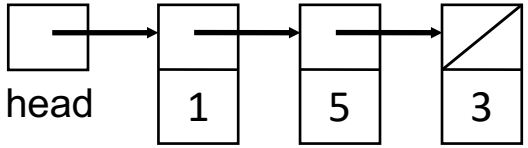
```
Node head = new Node(3,null);
```




```
head = new Node(5,head);
```




```
head = new Node(1,head);
```



- die drei oberen Anweisungen lassen sich auch in eine Anweisung zusammenfassen:

```
head = new Node(1,
       new Node(5,
       new Node(3,null)));
```

Liste durch Schleife aufbauen

```
Scanner in = new Scanner(System.in);

Node head = null;

System.out.println("Eingabe: ");

while (in.hasNextInt()) {
    int x  = in.nextInt();
    head = new Node(x,head);
}
```

Liste durchlaufen

Liste ausgeben:

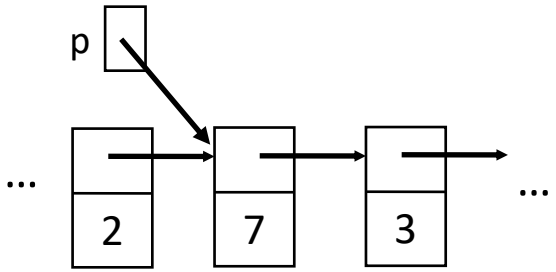
```
for (Node p = head; p != null; p = p.next) {
    System.out.println(p.data);
}
```

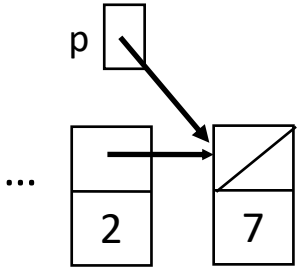
Liste nach x durchsuchen:

```
Node p;
for (p = head; p != null; p = p.next) {
    if (p.data == x) break;
}

if (p != null)
    System.out.println(x + " gefunden");
else
    System.out.println(x + " nicht gefunden");
```

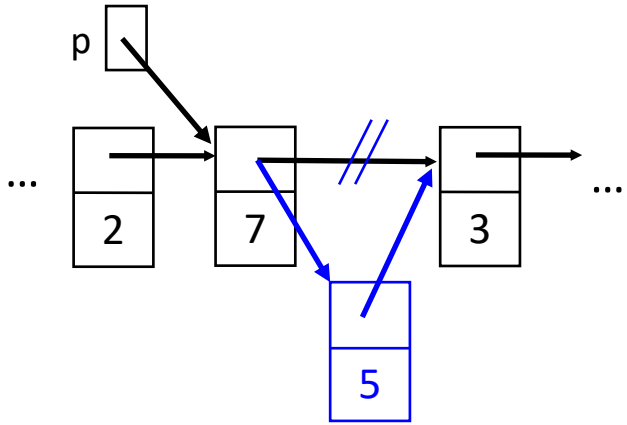
Einfügen nach einem beliebigen Knoten

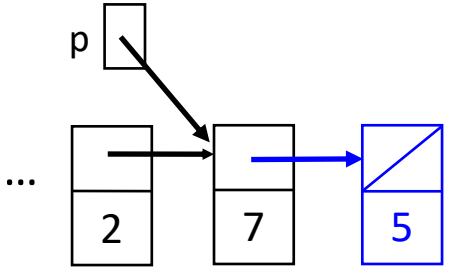
- Referenz p auf den Knoten positionieren, nach dem eingefügt werden soll.





- Einfügen nach p:

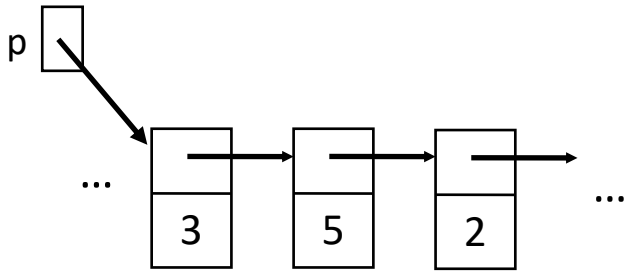


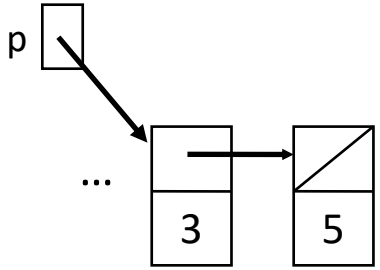



```
p.next = new Node(5,p.next);
```

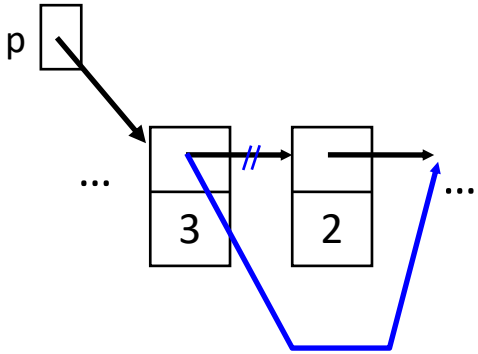
Löschen nach einem beliebigen Knoten

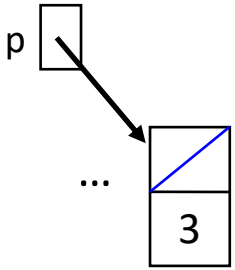
- Referenz p auf den Knoten positionieren, nach dem ein Knoten gelöscht werden soll. Es muss daher p.next != null gelten.





- Löschen des Knotens nach p:

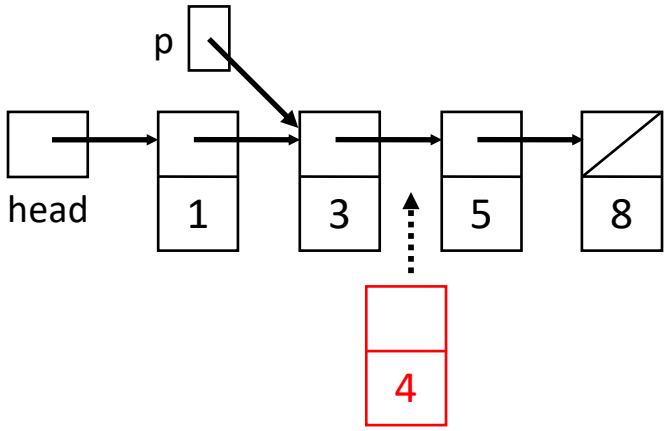




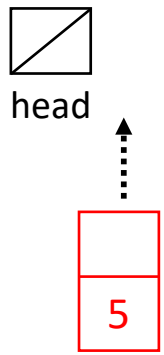

```
p.next = p.next.next;
```

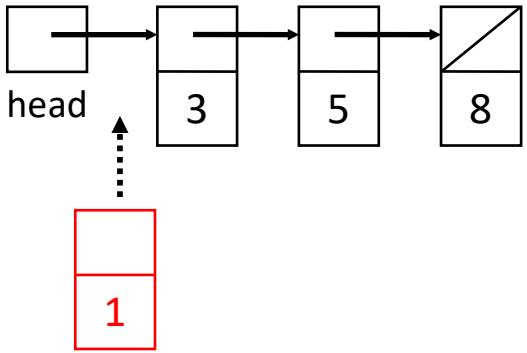
Einfügen in sortierte Liste (1)

- Allgemeiner Fall beim Einfügen von x:



  - Sonderfälle beim Einfügen von x:
    - Einfügen in leere Liste (d.h. head == null)
    - Einfügen am Anfang (d.h. x <= head.data)





Einfügen in sortierte Liste (2)

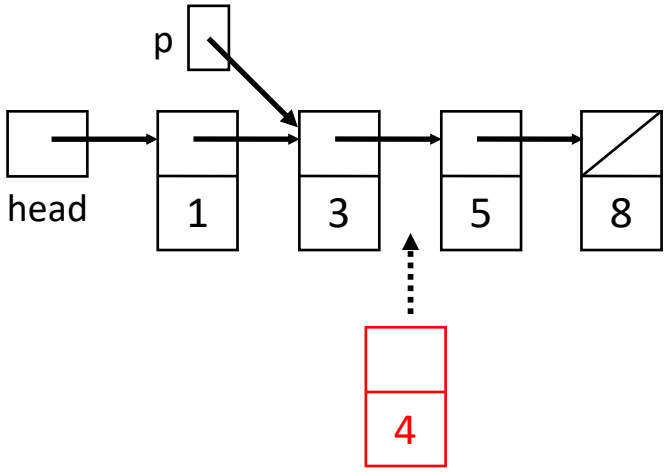
```
// Einfügen von x in sortierte Liste head:

if (head == null || x <= head.data) {
    head = new Node(x,head);
}
else {
    Node p = head;
    while(p.next != null && p.next.data < x) {
        p = p.next;
    }
    p.next = new Node(x,p.next);
}
```

Sonderfälle:

- Einfügen in leere Liste
- Einfügen am Listenanfang

Allgemeiner Fall:

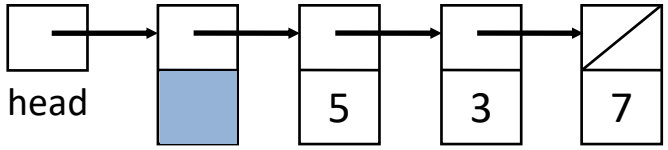
- Einfügen nach einem Knoten



Listen mit Hilfskopfknoten

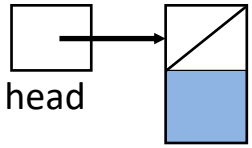
- Am Anfang der Liste wird zusätzlich ein nicht-datenspeichernder Knoten (Hilfskopfknoten; engl. dummy head node) eingefügt.
- Die leere Liste besteht damit genau aus einem Hilfskopfknoten.
- Vorteil: In der Regel keine Sonderfälle bei leerer Liste und Behandlung des ersten Datenknotens.

Liste mit einem Hilfskopfknoten und 3 Daten-Knoten:



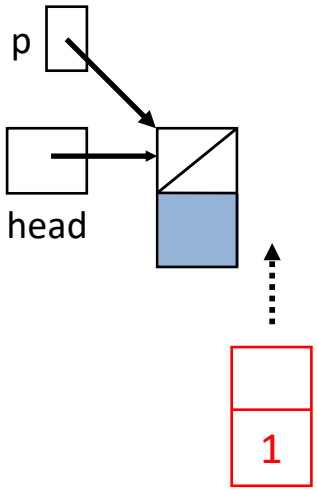
Leere Liste:

Enthält keinen Daten-Knoten.

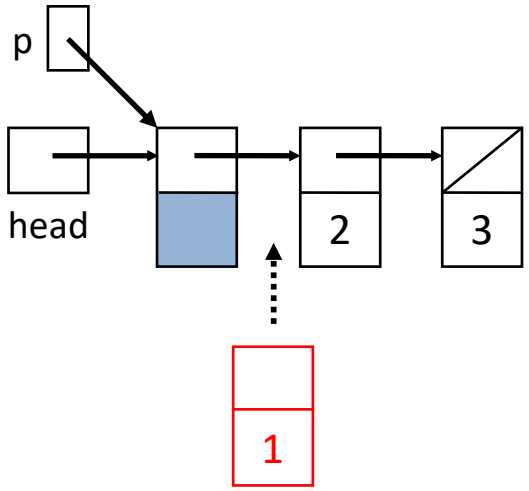


Einfügen in sortierte Liste mit Hilfskopfknoten

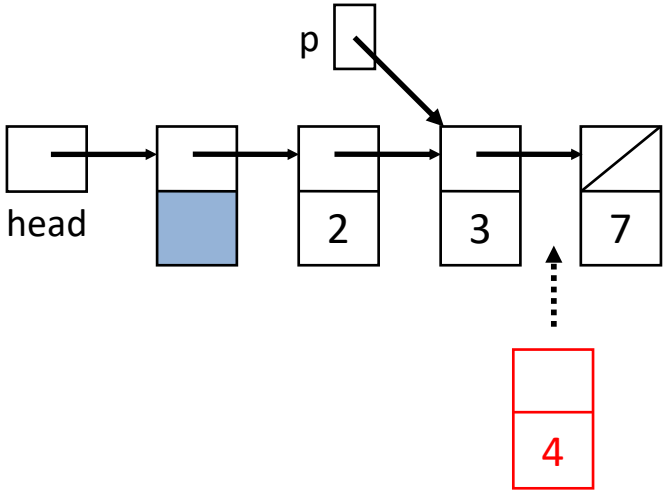
```
// Einfügen von x in sortierte Liste head:

Node p = head;
while(p.next != null && p.next.data < x) {
    p = p.next;
}
p.next = new Node(x,p.next);
```



Einfügen in leere Liste



Einfügen am Listenanfang



Einfügen nach einem Datenknoten

Kapitel 2: Datentyp Liste

- Einleitung
- Listen-Interface
- Liste als Feld: ArrayList
- Einfach verkettete Listen
- Hilfskopfknotentechnik
- Liste als einfach verkettete Liste: LinkedList
- Doppelt verkettete Listen

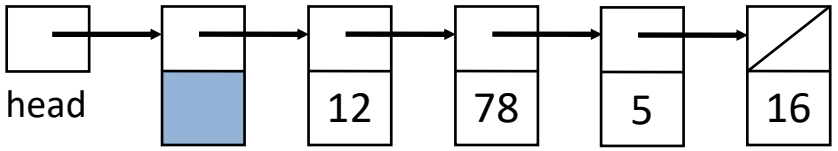
Datentyp Liste als einfach verkettete Liste

- Realisierung einer Liste als einfach verkettete Liste mit Hilfskopfknoten, um Sonderfälle bei den verschiedenen Operationen zu vermeiden.

Liste

- 12
- 78
- 5
- 16

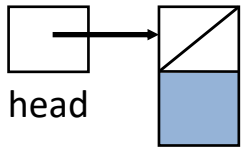
Liste als einfach verkettete Liste :



class LinkedList (1)

```
public class LinkedList implements List {

    public LinkedList() {clear();}

    public final void clear() {
        head = new Node(0,null);
        size = 0;
    }

    // ...

    private static class Node {
        Node next;
        int data;
        Node(int x, Node p) {
            data = x;
            next = p;
        }
    }
    private Node head;
    private int size;
}
```

Konstruktor legt leere Liste mit Hilfskopfknoten an.



Statisch geschachtelte Klasse Node: Verhält sich wie eine Top-Level-Klasse, jedoch hat LinkedList volle Zugriffsrechte auf Node.

Listenobjekt besteht aus head-Referenz für verkettete Liste und Anzahl der Listenelemente.

class LinkedList (2)

```
public class LinkedList implements List {
    // ...

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void add(int x) {
        add(size(), x);
    }

    public void add(int idx, int x) {
        if (idx < 0 || idx > size)
            throw new IndexOutOfBoundsException();
        Node p = head;
        for (int i = 0; i < idx; i++)
            p = p.next;
        p.next = new Node(x,p.next);
        size++;
    }
}
```

class LinkedList (3)

```
public class LinkedList implements List {

    // ...

    public int set(int idx, int x) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        Node p = head.next;
        for (int i = 0; i < idx; i++)
            p = p.next;
        int old = p.data;
        p.data = x;
        return old;
    }

    public int get(int idx) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        Node p = head.next;
        for (int i = 0; i < idx; i++)
            p = p.next;
        return p.data;
    }
}
```

class LinkedList (4)

```
public class LinkedList implements List {

    // ...

    public void remove(int idx) {
        if (idx < 0 || idx >= size)
            throw new IndexOutOfBoundsException();
        Node p = head;
        for (int i = 0; i < idx; i++)
            p = p.next;
        p.next = p.next.next;
        size--;
    }

    @Override
    public String toString() {
        StringBuilder s = new StringBuilder("");
        for (Node p = head.next; p != null; p = p.next) {
            s.append(p.data).append(", ");
        }
        s.append("size = ").append(size);
        return s.toString();
    }
}
```

class LinkedList (5)

```
public static void main(String[] args) {
     List l = new LinkedList();
     l.add(5);
     l.add(3);
     l.add(2);
     l.add(0,12);
     System.out.println(l);

     System.out.println(l.get(1));
     System.out.println(l.set(1,13));
     System.out.println(l.get(1));
     System.out.println(l);

     l.remove(1);
     System.out.println(l);
 }
```

```
12, 5, 3, 2, size = 4
```

```
5
5
13
12, 13, 3, 2, size = 4
```

```
12, 3, 2, size = 3
```

Aufgaben zu LinkedList

Aufgabe 2.3

Erweitern Sie die Klasse `LinkedList` um eine Methode `removeElement(int x)`, die aus der Liste das erste Element mit dem Wert x löscht.

Aufgabe 2.4

Erweitern Sie die Klasse `LinkedList` um eine Methode `removeAllElement(int x)`, die aus der Liste alle Elemente mit dem Wert x löscht.

Aufgabe 2.5

Erweitern Sie die Klasse `LinkedList` um eine Methode `append(List list)`, die eine weitere Liste list anhängt.

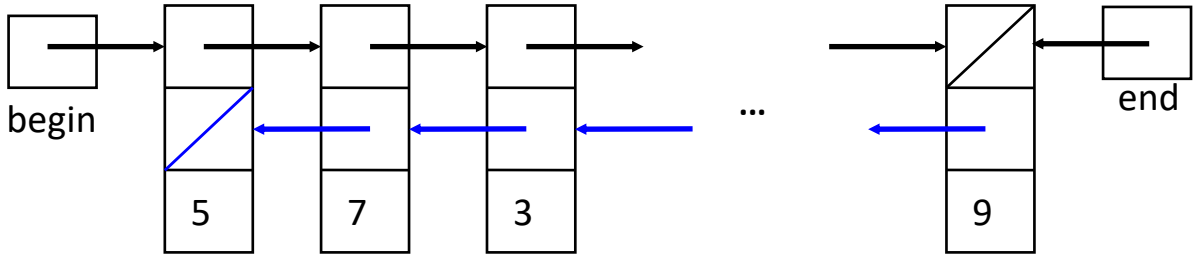
Aufgabe 2.6

Realisieren Sie einen Konstruktor `LinkedList(List list)`, der die verkettete Liste mit list initialisiert.

Kapitel 2: Datentyp Liste

- Einleitung
- Listen-Interface
- Liste als Feld: ArrayList
- Einfach verkettete Listen
- Hilfskopfknotentechnik
- Liste als einfach verkettete Liste: LinkedList
- Doppelt verkettete Listen

Doppelt verkettete Liste

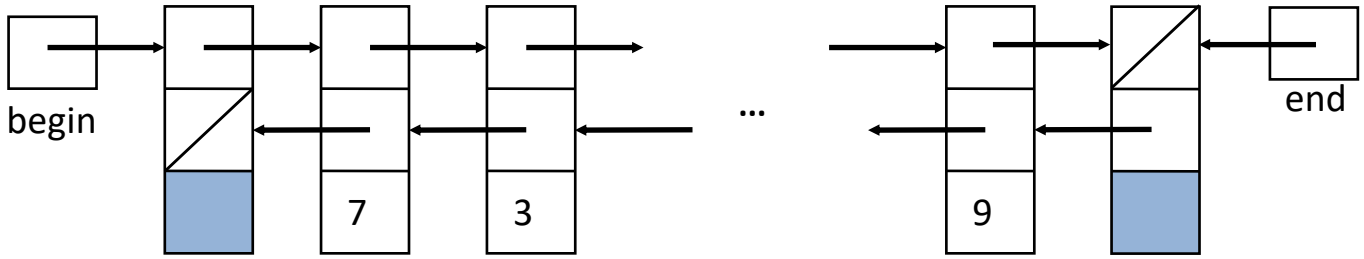


- Jeder Knoten enthält zusätzlich einen Zeiger auf den Vorgängerknoten.
- Damit effizienter Zugriff auf Vorgängerknoten möglich.
- Oft ist auch Speicherung einer Referenz auf den letzten Knoten geschickt.
- Andere Bezeichnungen: begin als Referenz auf den ersten Knoten und end als Referenz auf den letzten Knoten.

```
class Node {

    Node next;
    Node prev; // previous
    int data;

    Node(int x, Node n, Node p) {
        data = x;
        next = n;
        prev = p;
    }
}
```

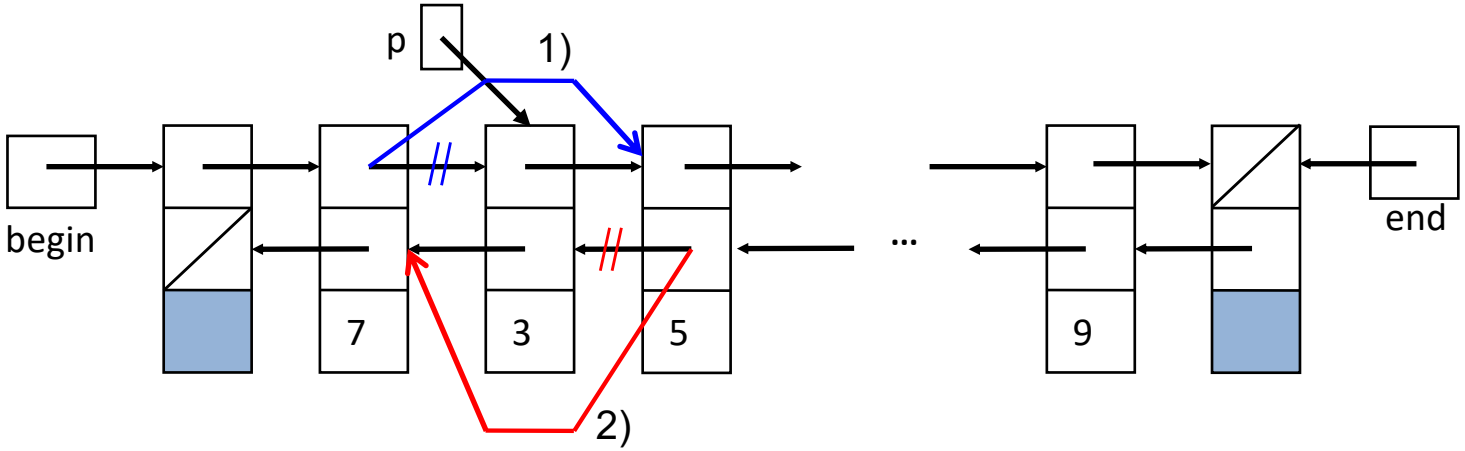
Doppelt verkettete Liste mit Hilfsknoten (1)

- Durch zusätzliche nicht datenspeichernde Hilfknoten am Anfang und am Ende werden Behandlung von Spezialfällen überflüssig.



- Löschen eines Daten-Knotens p:

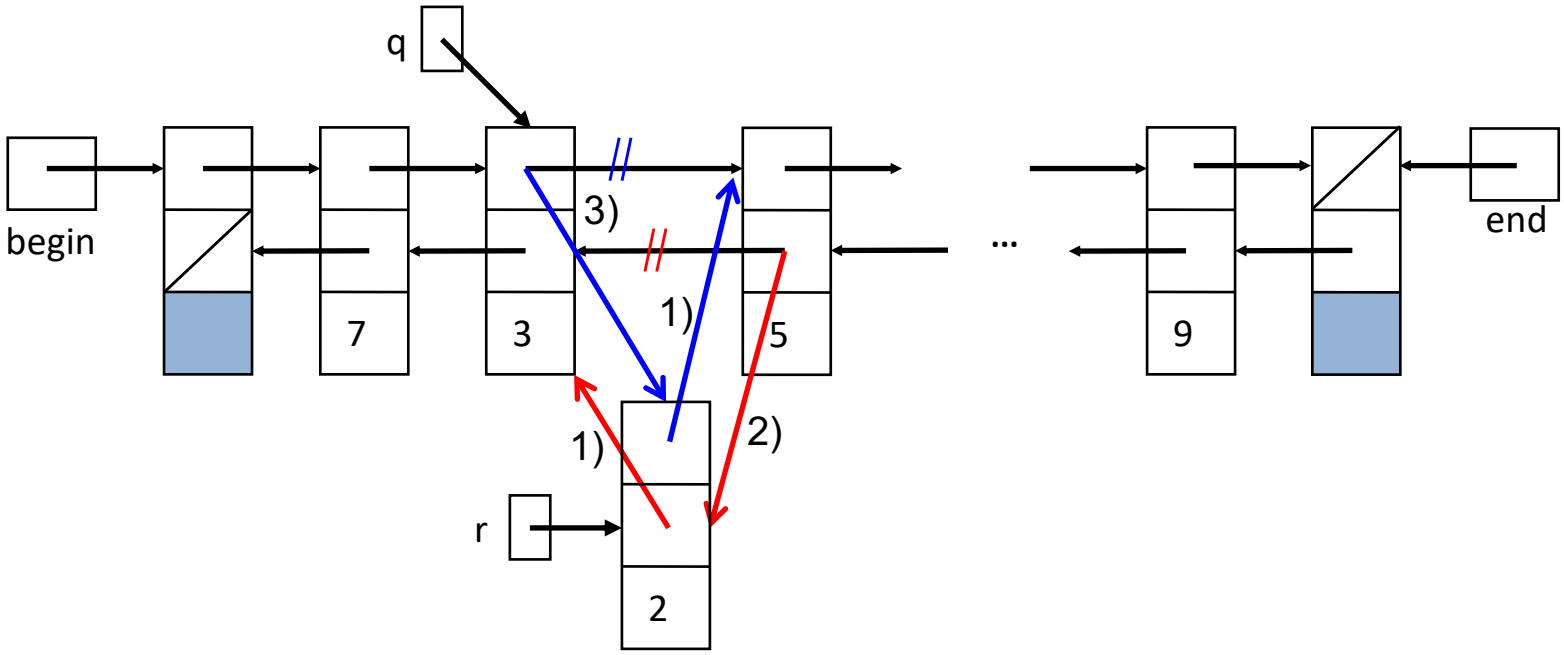
```
p.prev.next = p.next;      // 1)
p.next.prev = p.prev;      // 2)
```



Doppelt verkettete Liste mit Hilfsknoten (2)

- Einfügen eines neuen Knotens r nach einem Knoten q:

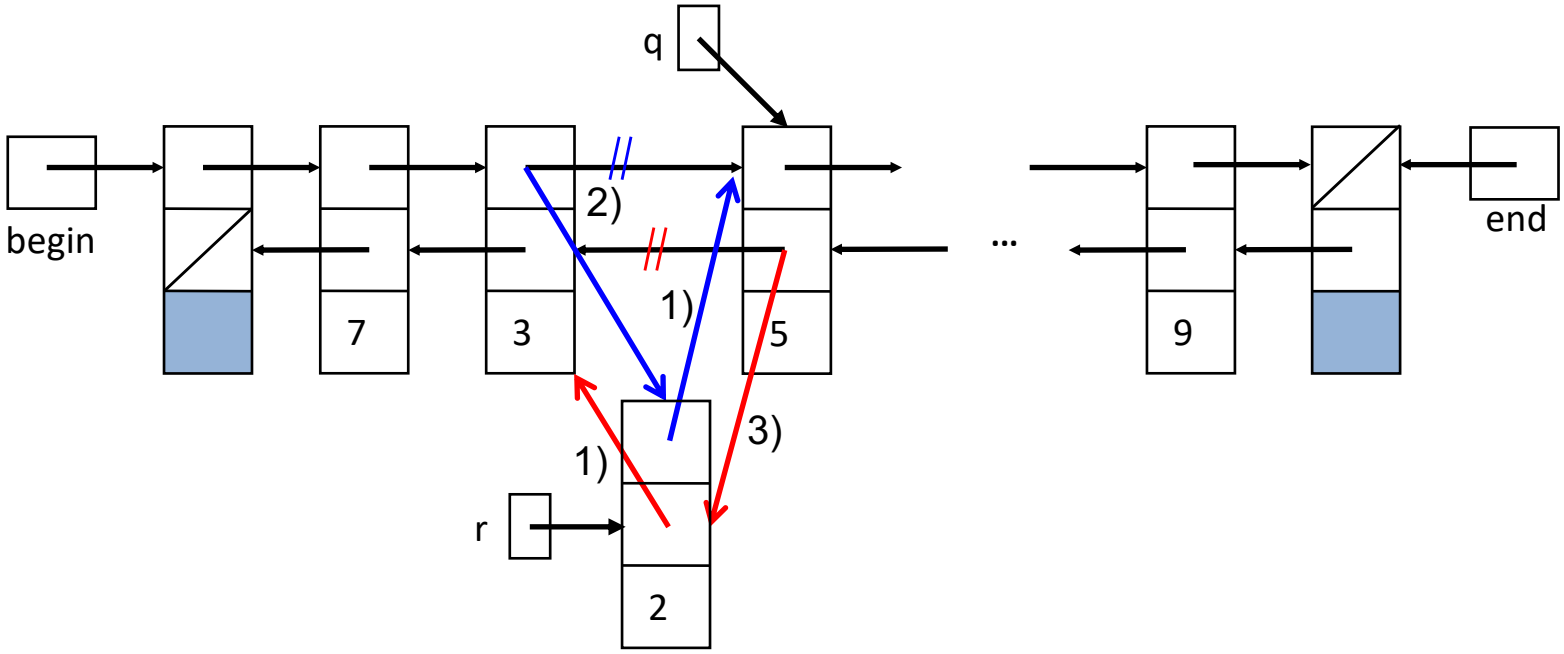
```
Node r = new Node(x, q.next, q);      // 1)
r.next.prev = r;                      // 2)
q.next = r;                           // 3)
```



Doppelt verkettete Liste mit Hilfsknoten (3)

- Einfügen eines neuen Knotens r vor einem Knoten q:

```
Node r = new Node(x, q, q.prev);      // 1)
r.prev.next = r;                      // 2)
q.prev = r;                           // 3)
```



Aufgaben zu doppelt verketteten Listen

Aufgabe 2.7

Der Typ Liste soll nun mit doppelt verketteten Listen mit Hilfskopfknoten realisiert werden.

- Definieren Sie die Methode clear().
- Definieren Sie die Methode add(i,x) so, dass bei höheren Indexwerten i die Liste von hinten nach vorne durchlaufen wird.

```
public class DoubleLinkedList
   implements List {

   private static class Node {
      private Node next;
      private Node prev; // previous
      private int data;

      public Node(int x, Node n, Node p){
         data = x;
         next = n;
         prev = p;
      }
   }

   private Node begin;
   private Node end;
   private int size;

   // ...
}
```