Überblick

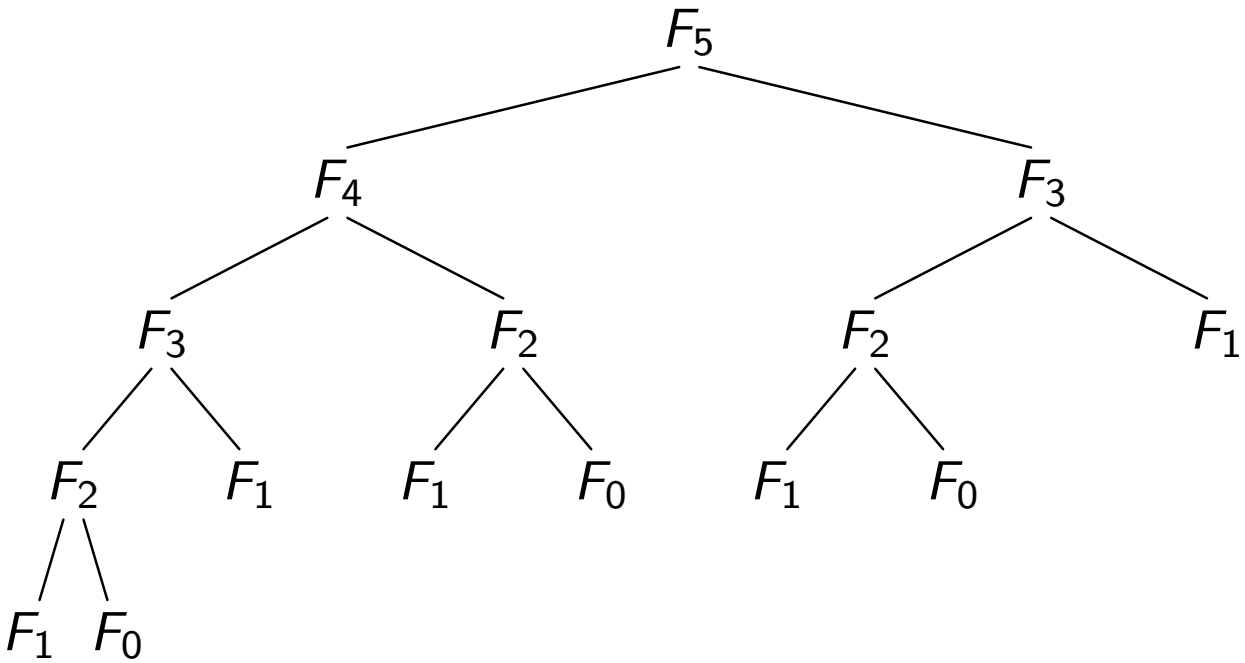
Nochmals Fibonacci-Zahlen

```
int fibo(int n) {
  if(n == 0) {
    return 0;
  }
  else if(n == 1) {
    return 1;
  }
  else {
    return fibo(n - 1) + fibo(n - 2);
  }
}
```

Wie schnell ist dieses Programm? Sehr langsam!

Nochmals Fibonacci-Zahlen

Problem: Gleiche Werte werden *mehrfach* berechnet.



Lösung: *Memorization*

In Wirklichkeit heißt es „Memoization“, aber immer mehr Leute verwenden „Memorization“. In naher Zukunft wird das also der korrekte Begriff durch Mehrheitsmeinung werden...

Memorization

Speichere neu berechnete Werte.

Verwende gespeicherte Werte, wenn vorhanden.

```
long fibo2(int n) {
  if(n == 0) {
    return 0;
  }
  else if(n == 1) {
    return 1;
  }
  if(f[n] == 0) {
    f[n] = fibo2(n - 1) + fibo2(n - 2);
  }
  return f[n];
}
```

Überblick

2. Rekursion
   - Rekursive Methoden
   - Backtracking
   - Memorization
   - Einfache rekursive Datenstrukturen
   - Bäume
   - Aufzählen, Untermengen, Permutationen, Bitmengen

Rekursive Datenstrukturen

Wir möchten eine *Liste* bzw. eine *endliche Folge* implementieren.

Operationen:

- **int** *head*(): Was ist das erste Element?
- *Liste* *tail*(): Die Liste ohne das erste Element.
- **boolean** *isempty*(): Ist die Liste leer?
- *Liste* *add*(**int** *elem*): Dieselbe Liste, aber *elem* davor.

Rekursiver Entwurf:

Eine Liste ist entweder

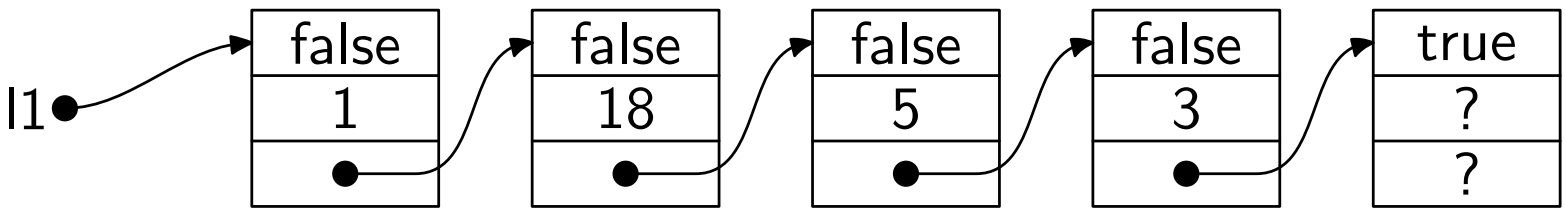
- leer oder
- besteht aus einem Element und einer Liste.

```
class Liste {
  private boolean empty; // Liste leer?
  private int value; // erstes Element
  private Liste rest; // restliche Elemente
   //
  int head() {...}
  Liste tail() {...}
  boolean isempty() {...}
   // Erzeuge neue Liste [elem, alte Liste]
  Liste add(int elem) {...}
}
```

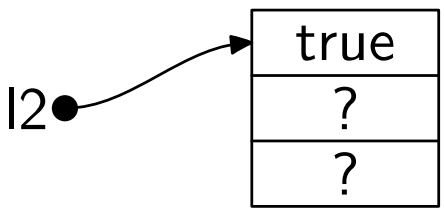
Eine *Liste* darf eine *Liste* „enthalten“!

Interne Repräsentation

Liste 1, 18, 5, 3:



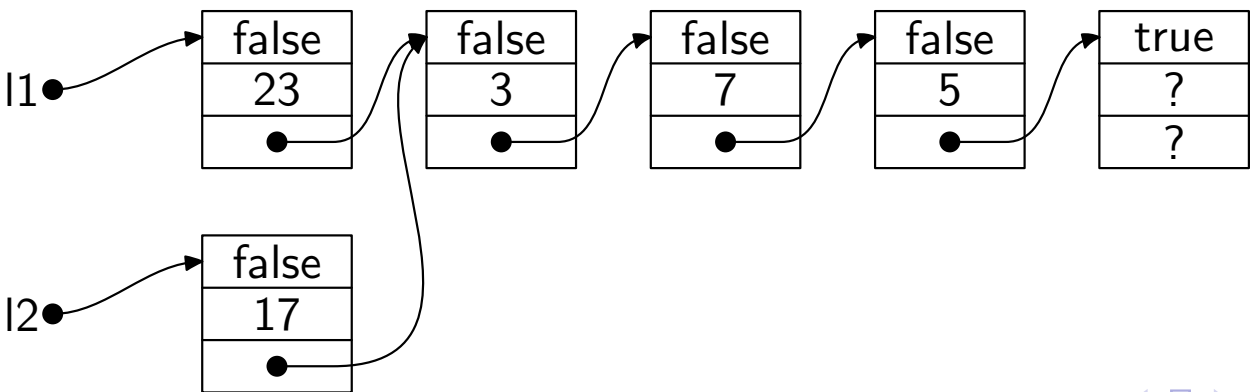
Leere Liste:



Beispiel

```
public static void main(String args[ ]) {
   Liste l1 = new Liste();
   l1 = l1.add(5);
   l1 = l1.add(7);
   l1 = l1.add(3);
   System.out.println(l1); // [3, 7, 5]
   Liste l2 = l1;
   l2 = l2.add(17);
   l1 = l1.add(23);
   System.out.println(l1);
   System.out.println(l2);
}
```

```
public static void main(String args[ ]) {
    Liste l1 = new Liste();
    l1 = l1.add(5);
    l1 = l1.add(7);
    l1 = l1.add(3);
    System.out.println(l1); // [3, 7, 5]
    Liste l2 = l1;
    l2 = l2.add(17);
    l1 = l1.add(23);
    System.out.println(l1);
    System.out.println(l2);
}
```



Reference- und Value-Semantik

```
Student karl = new Student("Karl","Ranseier","123456");
...
Student klausurTeilnehmer = karl;
...
karl.setVorname("Charles");
klausurTeilnehmer.getVorname(); // ergibt Charles
```

Sowohl *karl* als auch *klausurTeilnehmer* beziehen sich auf dieselbe Person.

Bei einer Zuweisung wird keine Kopie angefertigt.

Dieses Verhalten ist natürlich für eine Klasse *Student*.

Ein Algorithmus, wie er in Lehrbüchern steht:

procedure *Dijkstra*(s) :
$Q := V - \{s\}$;
for $v \in Q$ **do** $d[v] := \infty$ **od**;
$d[s] := 0$;
while $Q \mathrel{!}= \emptyset$ **do**
  choose $v \in Q$ *with minimal* $d[v]$;
  $Q := Q - \{v\}$;
  forall *u adjacent* **to** *v* **do**
    $d[u] := min\{d[u], d[v] + length(v, u)\}$
  od
od

Q und V sind Mengen.

In Java z.B. $V.subtract(s)$ statt $V - \{s\}$.

Bei Mengen erwarten wir eine Value-Semantik.

Immutable Klassen

Es seien A, B, C Mengen.

$A := \{1, 2, 3\}$
$B := A$
$A := A \cup \{4\}$

Mit Mengen sind wir gewohnt so zu arbeiten.

Wir erwarten, daß $B = \{1, 2, 3\}$ und $A = \{1, 2, 3, 4\}$.

Dieses Verhalten können wir durch immutable Klassen erreichen.

Immutable Klassen

Liste ist immutable:

- Es gibt keine Methode, die ein Objekt des Typs *Liste* ändern kann.
- Auf die Instanzvariablen kann man nicht direkt zugreifen.

Daher verhält sich *Liste* genauso wie **int** und **double**.

```
int n = 5;
int k = n + 3; // n ist immer noch 5
Liste list = new Liste();
list = list.add(5);
list = list.add(3); // list ist [3, 5]
Liste lang = list.add(1); // list bleibt [3, 5]
```

Immutable Klassen

Wollen wir eine Klasse *immutable* machen, dann können wir alle Instanzvariablen **final** deklarieren:

```
class Liste {
  private final boolean empty; // ist diese Liste leer?
  private final int value; // das erste Element dieser Liste
  private final Liste rest; // restliche Liste ohne erstes Element
```

Eine **final**-Variable kann nur in einem Konstruktor verändert werden.

So ist die immutable-Eigenschaft garantiert.

Überblick

Weitere rekursive Datenstrukturen

Ein *Baum* ist eine Wurzel, die mehrere Kinder haben kann, welche selbst Bäume sind. Einen Baum ohne Kinder nennt man *Blatt*.

```
public class Tree {
  private int root; // Ein Baum aus Zahlen
  private List<Tree> children;
  public Tree(int r) {
    root = r;
    children = new List<Tree>();
  }
  public List<Tree> getChildren() {return children;}
  public int getRoot() {return root;}
  public void addChild(Tree newChild) {
    children = children.add(newChild);
  }
```

Was ist *List⟨Tree⟩* genau?

```
class List⟨T⟩ {
  private final boolean empty; // ist diese Liste leer?
  private final T value; // das erste Element dieser Liste
  private final List⟨T⟩ rest; // restliche Liste ohne erstes Element
  public List() { // erzeuge eine neue leere Liste
    empty = true;
    value = null;
    rest = null;
  }
  private List(T elem, List⟨T⟩ rest) {
    this.empty = false;
    this.value = elem;
    this.rest = rest;
  }
  public boolean isempty() {
    return empty;
  }
```

Generische Klassen

List⟨*Tree*⟩ verwendet die *generische Klasse* **class** *List*⟨*T*⟩.

Eine generische Klasse hat einen zusätzlichen *Typparameter*.

Auf diese Weise können wir z.B. folgendes schreiben:

```
List<Person> freunde = new List<Person>();
Person karl = new Student("Karl","Ranseier","123456");
Person joe = new Person("Joe","Miller");
freunde.add(karl);
freunde.add(joe);
```

So haben wir *Liste* sehr verallgemeinert!