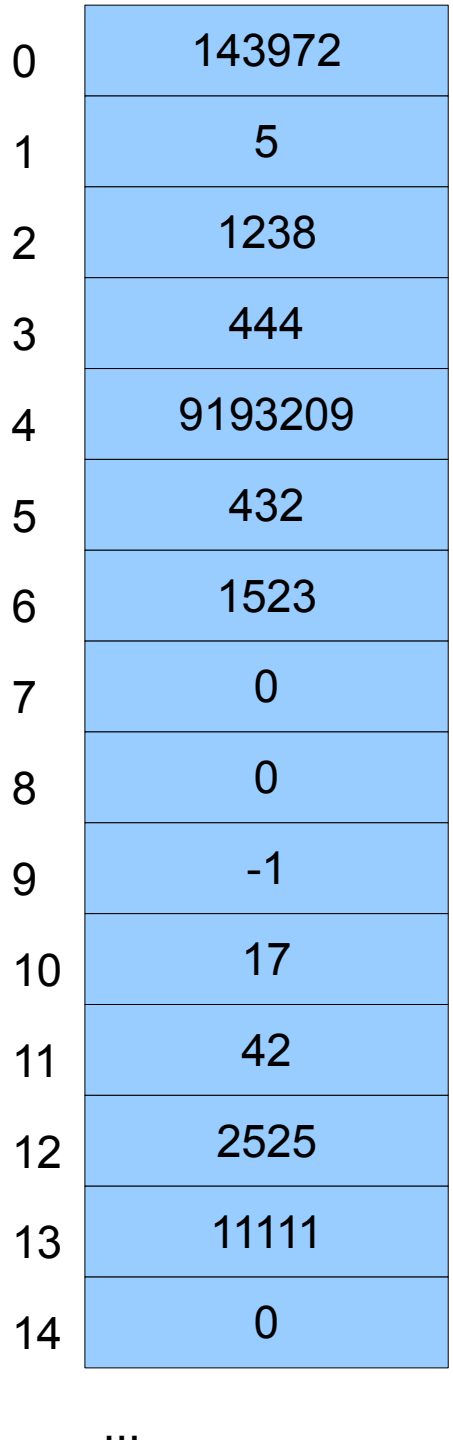
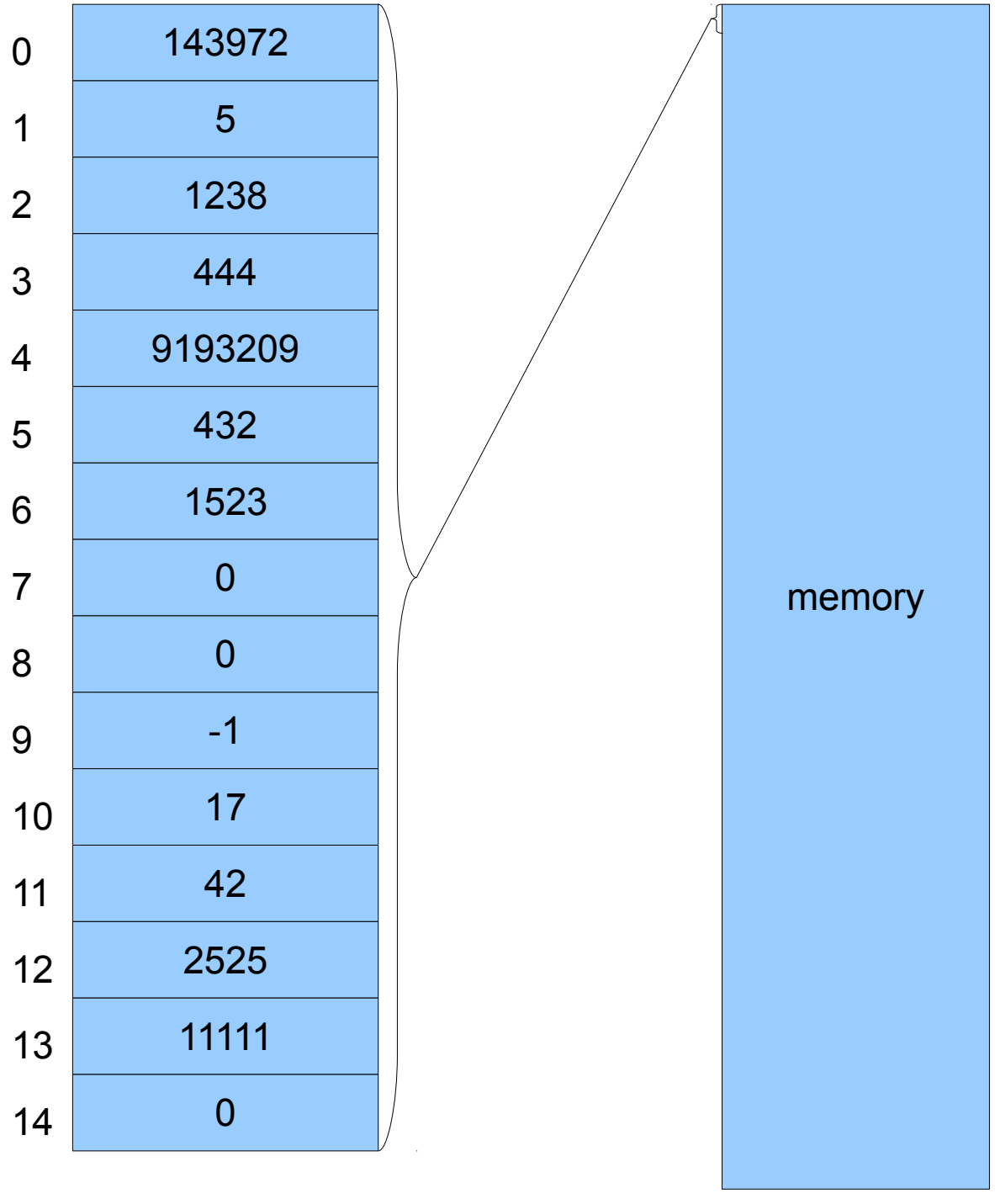
Abschnitt 4.2

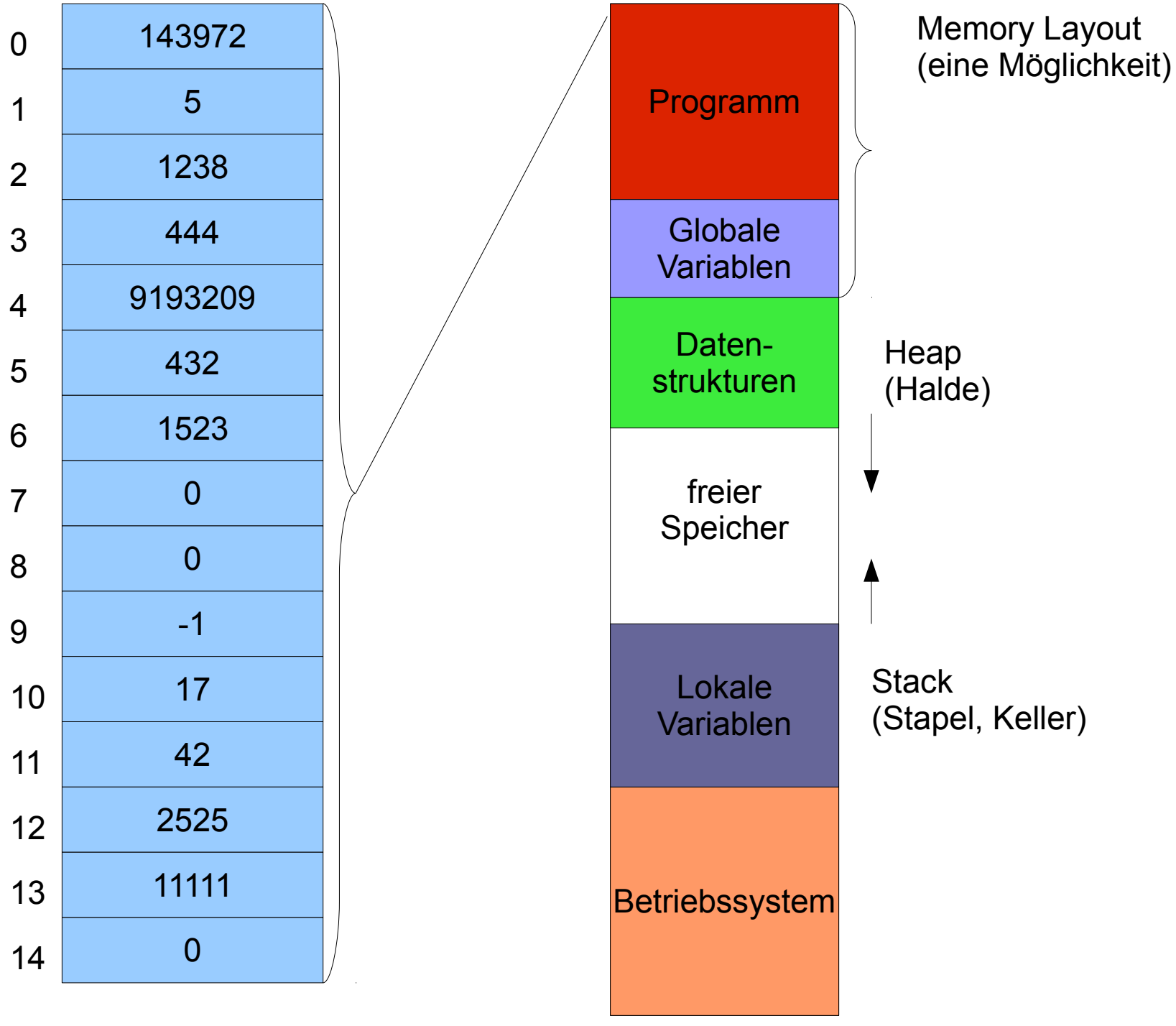
Speicher und Datenstrukturen

Unterabschnitt 4.2.1

Speicherorganisation und CPU


0
143972
1
5
2
1238
3
444
4
9193209
5
432
6
1523
7
0
8
0
9
-1
10
17
11
42
12
2525
13
11111
14
0
...


0
143972
1
5
2
1238
3
444
4
9193209
5
432
6
1523
7
0
8
0
9
-1
10
17
11
42
12
2525
13
11111
14
0
...
memory


0
143972
1
5
2
1238
3
444
4
9193209
5
432
6
1523
7
0
8
0
9
-1
10
17
11
42
12
2525
13
11111
14
0
...
Programm
Globale Variablen
Daten-strukturen
freier Speicher
Lokale Variablen
Betriebssystem
Memory Layout
(eine Möglichkeit)
Heap
(Halde)
Stack
(Stapel, Keller)

CPU und Speicher

```
int[] memory = new int[1000000];

load(memory,"boot-image");

int pc = 0;
int a = 0;

for(;;) {
  int instruction = memory[pc];
  switch(instruction) {
  case 0: pc++; break;
  case 1: a = 0; pc++; break;
  case 2: a = memory[pc+1]; pc += 2; break;
  case 3: a = memory[memory[pc+1]]; pc += 2; break;
  case 4: System.out.print((char)a); System.out.flush(); pc++; break;
  // many more cases
  }
}
```

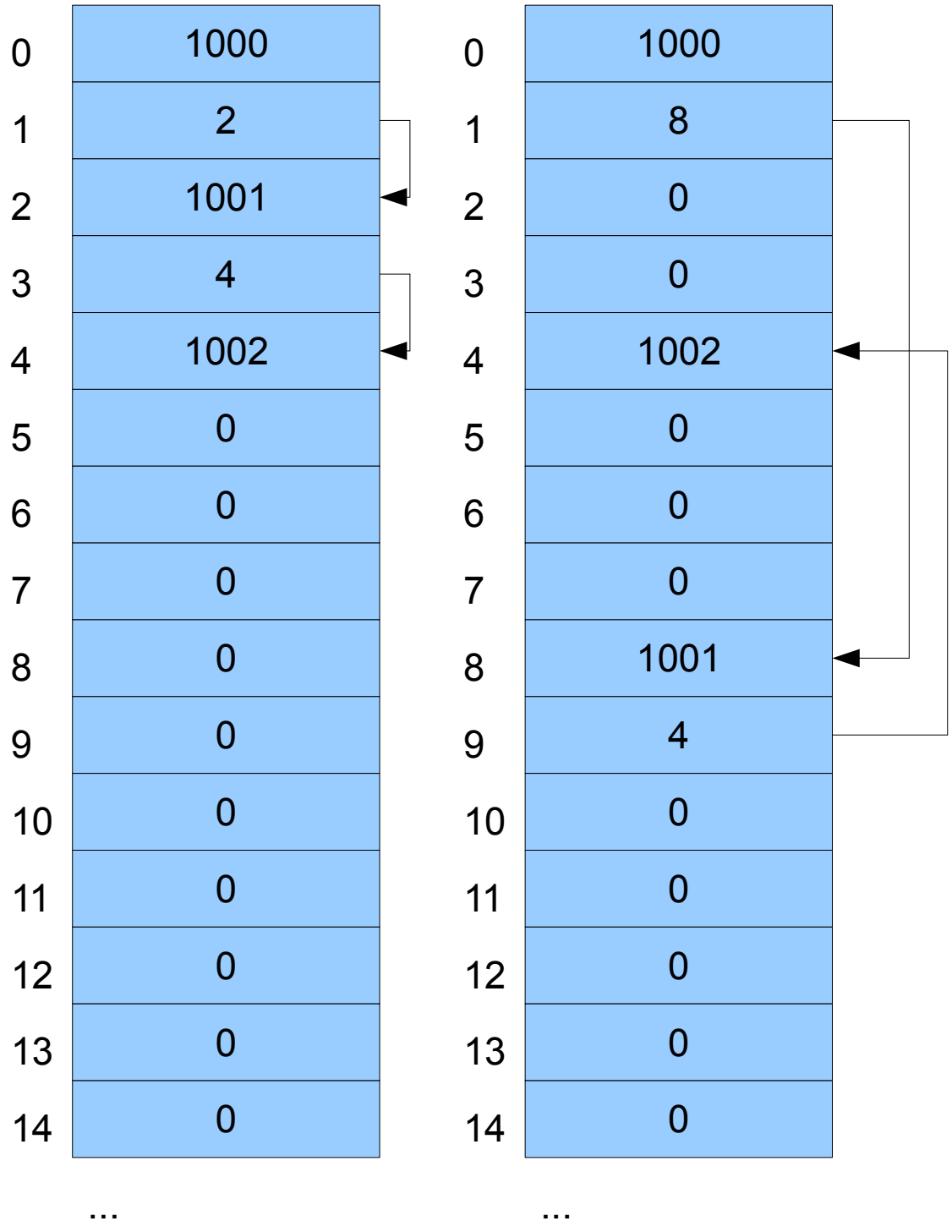
Unterabschnitt 4.2.2

Boxen und Pfeile

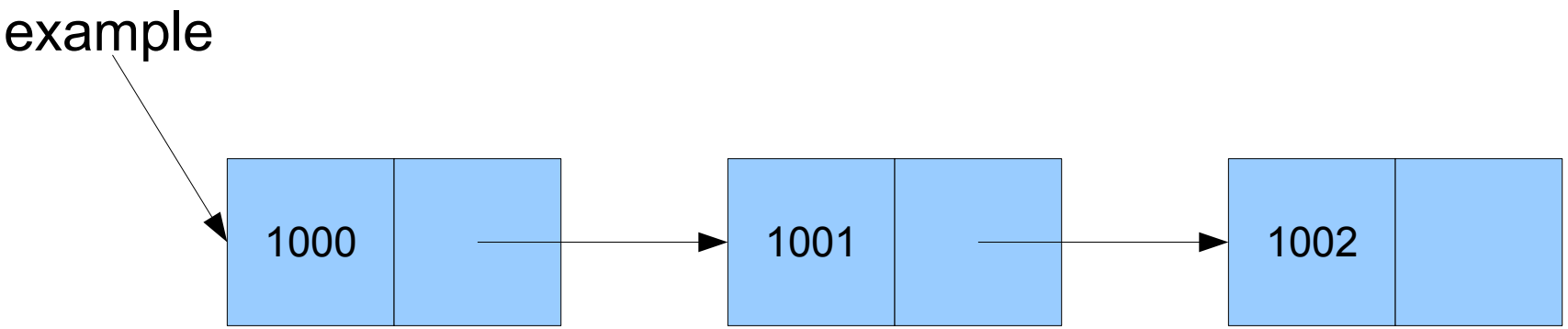
Listen

```
class List {
  int head;
  List tail;
  static List cons(int head,List tail) {
    List l = new List();
    l.head = head;
    l.tail = tail;
  }
  static example = cons(1000,cons(1001,cons(1002,null)));
}
```

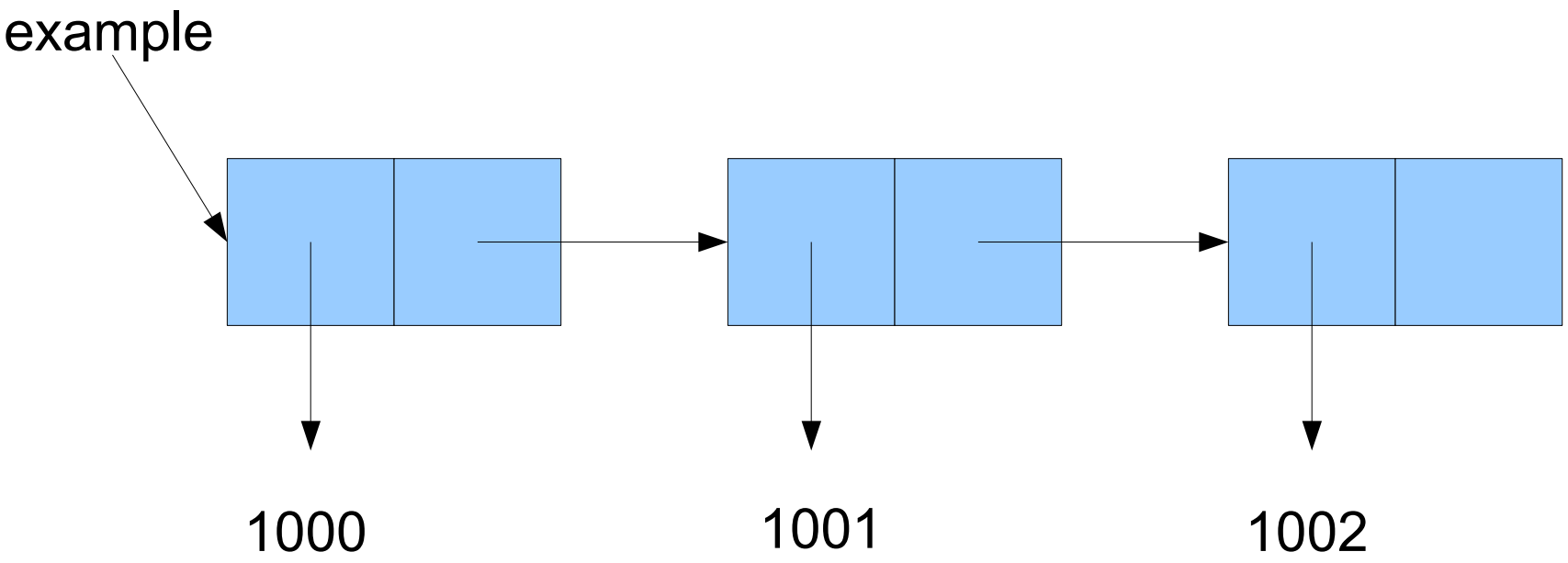
[1000,1001,1002]



[1000,1001,1002]



[1000,1001,1002]



Datenstrukturen und Speicher

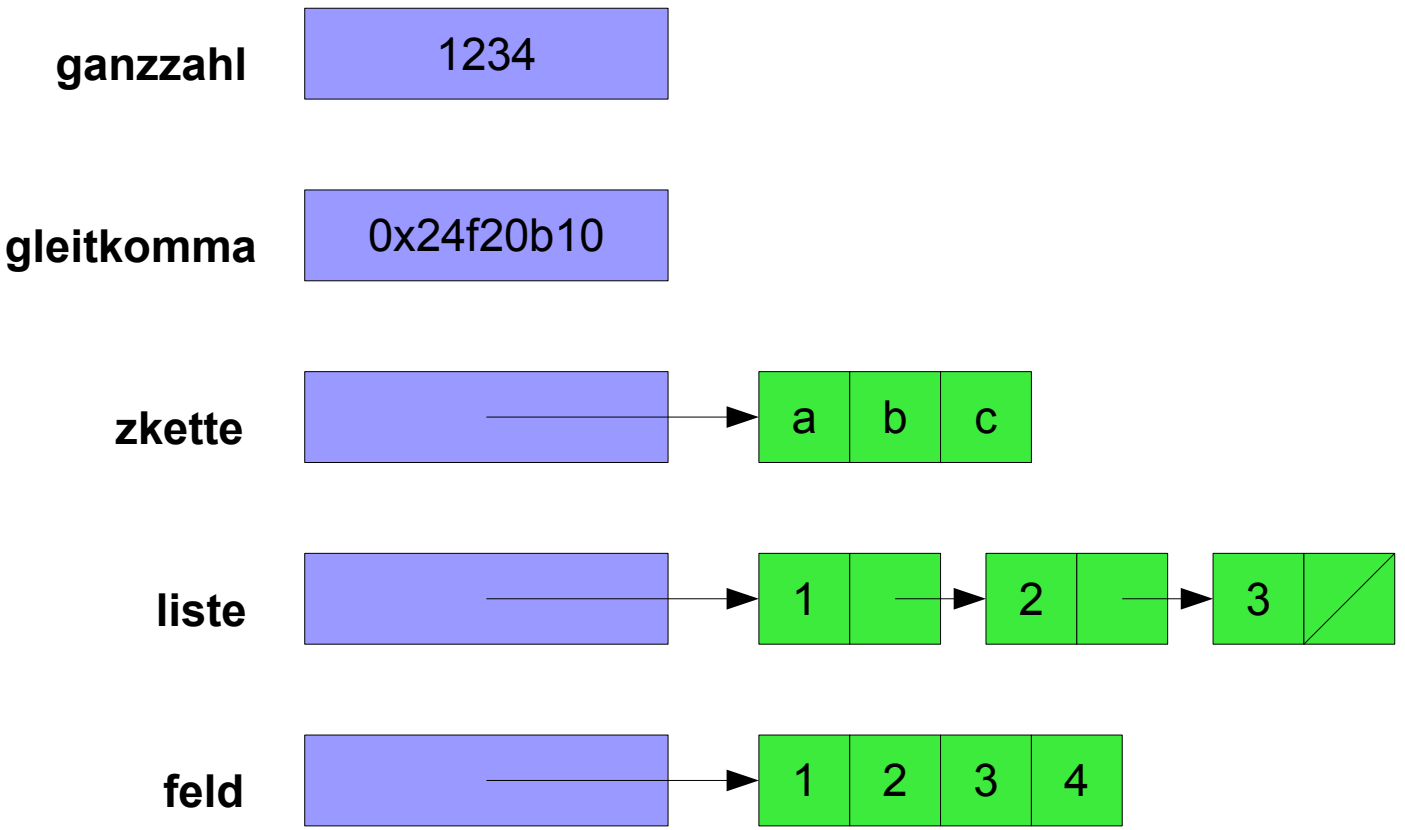
- der Speicher ist eine nummerierte Folge von Worten
- jedes Wort kann eine Ganzzahl speichern
- Worte können als numerische Werte oder als Zeiger interpretiert werden
- Interpretation hängt vom Programm ab
- Visualisierung von Speicherinhalten durch Boxen, Pfeile und Werte
  - Boxen = Speicherzellen
  - Pfeile = Interpretation von Worten als Zeiger
  - Werte = Zahlen in Box

Unterabschnitt 4.2.3

Variablen, Datenstrukturen, Speicher

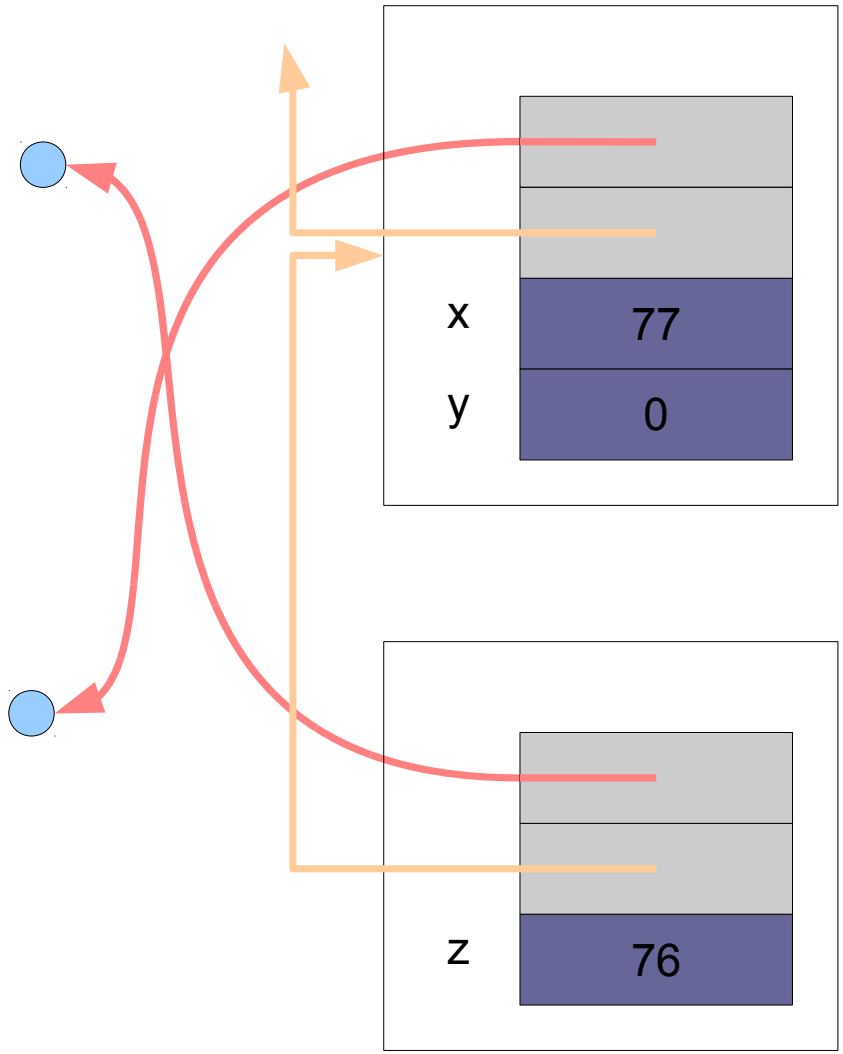
Globale Variablen

```
int         ganzzahl = 1234;
float       gleitkomma = 1.298324e20;
String      zkette = “abc”;
List  list = cons(1,cons(2,cons(3,null)));
int[] feld = {1,2,3,4};
```



Lokale Variablen

```
int f(int x) {
  int y = g(x-1);
  return 2*y;
}

int g(int z) {
  return -z;
}

f(77);
```

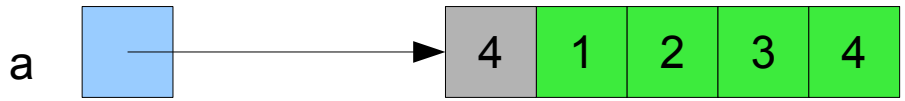


Variablen

- globale Variablen sind an festen Adressen bei Programmstart
- lokale Variablen werden in speziellen internen Datenstrukturen abgelegt
- Datenstrukturen des Benutzers werden in einem separaten Bereich angelegt

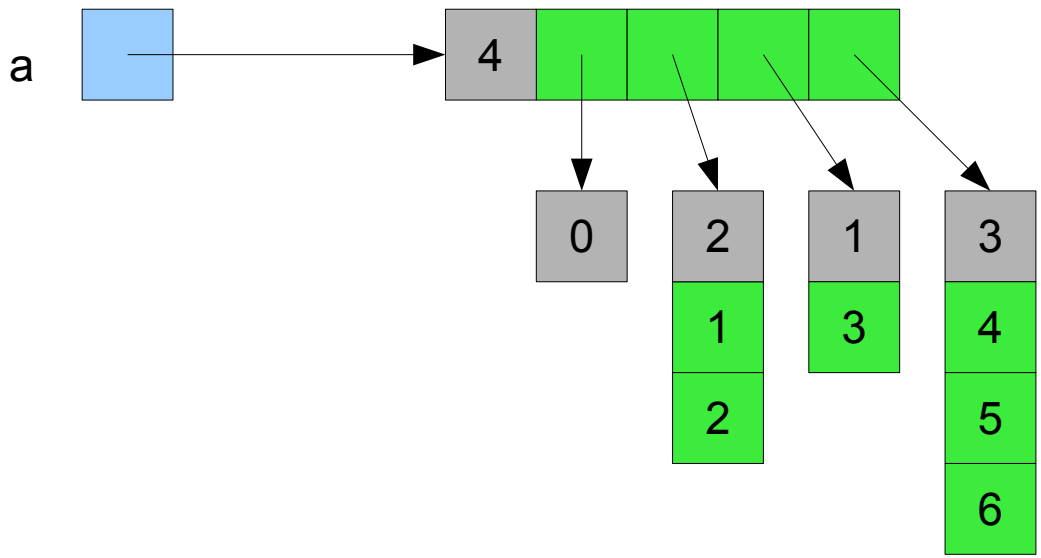
Feld / Array

```
int[] a = {1,2,3,4};
```



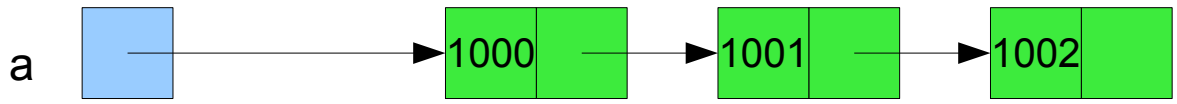
Ragged 2D Array

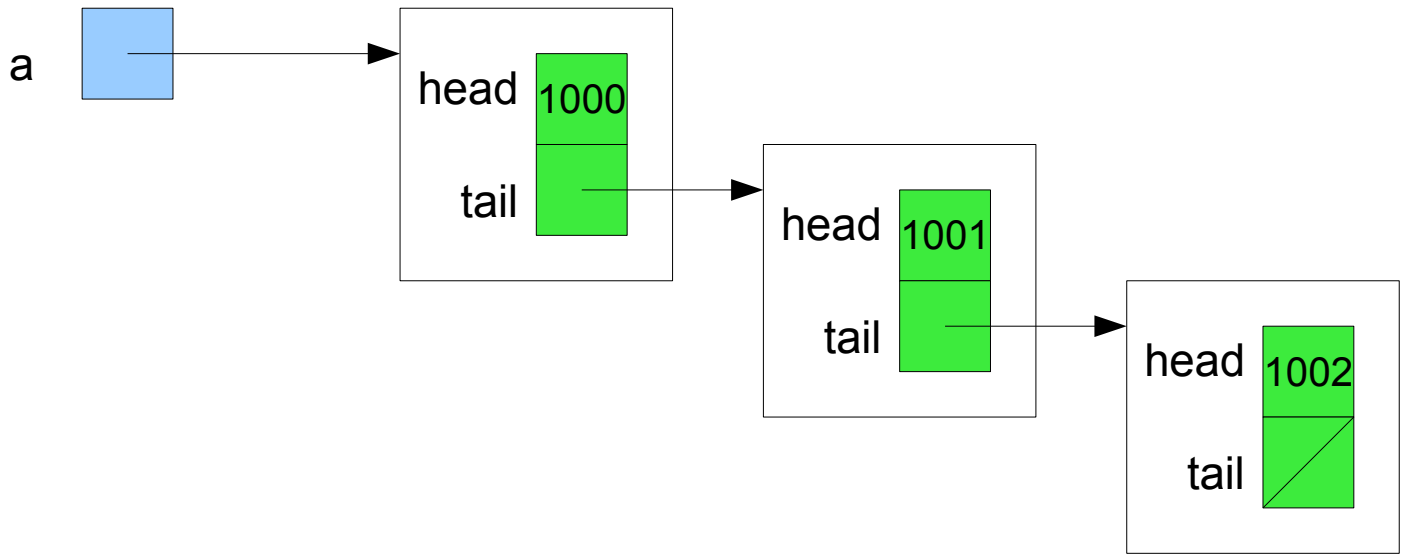
```
int[][] a = {{},{1,2},{3},{4,5,6}};
```



[1000,1001,1002]

```
class List {
  int head;
  List tail;
}
```





Verbundtypen und Felder

- Verbundtypen und Felder werden in Java immer im Heap angelegt
- Feld = Länge + variable Anzahl von Worten für Elemente
- Verbundtyp = Feste Anzahl von Worten, entsprechend der Komponenten
- In Java: Komponentennamen werden nicht explizit gespeichert

Unterabschnitt 4.2.4

Prozeduraufrufe

Prozeduraufruf

- erzeuge einen neuen Stackframe (eine “Inkarnation”)
- speichere die Adresse des vorherigen Stackframes
- speichere den gegenwärtigen Programmschritt (“PC”, “Zeile”)
- erzeuge genügend Raum für die Argumente
- speichere die Argumente im Stackframe
- Springe zum Anfang der Prozedur
- erzeuge weiteren Raum für die lokalen Variablen
- initialisiere die lokalen Variablen
- führe die Prozedur aus
- lade den gespeicherten Schritt und kehre zurück
- lösche den Stackframe

Prozeduraufruf (2)

```
void f() {
  g();
}
void g() {
  h();
}
void h() {
  print("hello");
}
```

- erzeuge Inkarnation von f
- erzeuge Inkarnation von g
- erzeuge Inkarnation von h
- zerstöre Inkarnation von h
- zerstöre Inkarnation von g
- zerstöre Inkarnation von f

Prozeduraufruf (3)

- In Java und C sind Funktionsaufrufe immer “geklammert”.
  Dies erlaubt die Verwendung eines Stapelspeichers (LIFO usw.) für Inkarnationen
- In Haskell können Inkarnationen als Werte behandelt werden.
  Dies erfordert komplexere Strategien.

Variablen in Programm und Speicher

Jede Variablendeklaration definiert eine ***Programmvariable***. Wir unterscheiden:

- globale Variablen
- (prozedur-)lokale Variablen
- Klassenvariablen, Attribute, usw.

Variablen in Programm und Speicher (2)

Jeder Programmvariablen entsprechen zur Ausführungszeit/Laufzeit des Programms eine oder mehrere Speichervariablen:

- Jeder globalen Variablen ist genau eine Speichervariable zugeordnet.
- Ist *v* eine lokale Variable zur Prozedur *p*, dann gibt es zu jeder Inkarnation von *p* eine Speichervariable für *v*.
- (andere Variablentypen behandeln wir später).

Definition: (Lebensdauer von Variablen)

Die ***Lebensdauer*** einer Speichervariable ist der Teil des Ablaufs, in dem sie für die Ausführung bereit steht.

Die Lebensdauer globaler Variablen erstreckt sich über den gesamten Ablauf des Programms.

Variablen zu lokalen Deklarationen leben solange wie die zugehörige Prozedurinkarnation bzw. über die Dauer der Ausführung ihres Blocks.

Bemerkung:
Es gibt auch Variablen, deren Lebensdauer direkt über Anweisungen gesteuert werden kann.

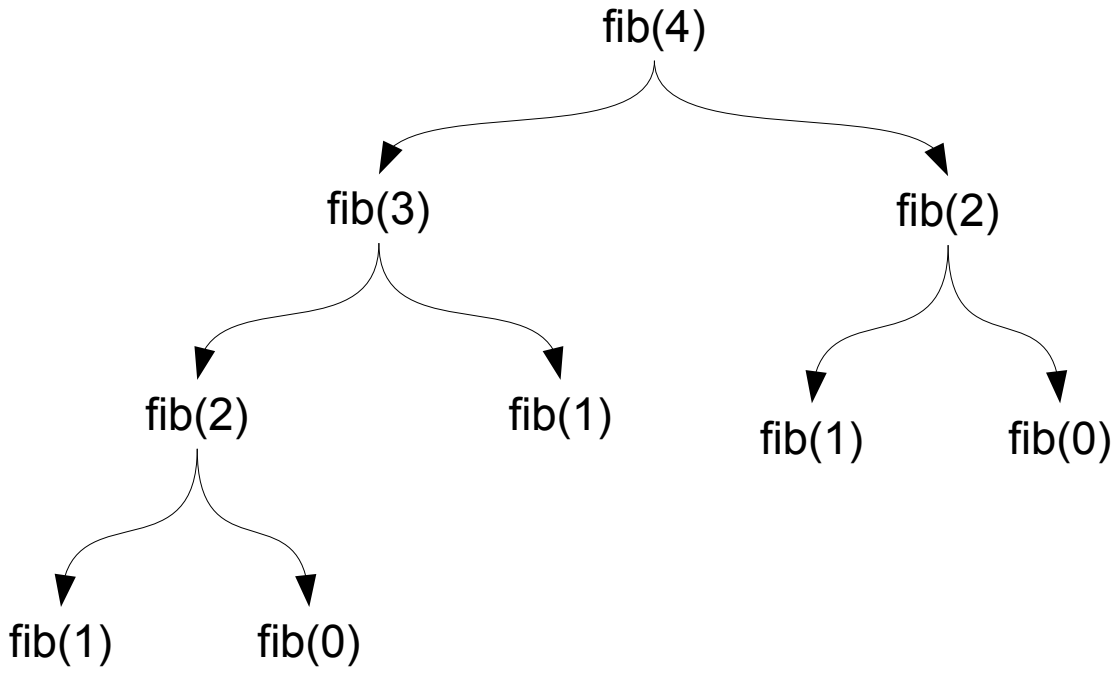
Prozedurbaum

Der ***Prozeduraufrufbaum*** beschreibt die Prozeduraufrufstruktur in einem Programm- oder Prozedurablauf.

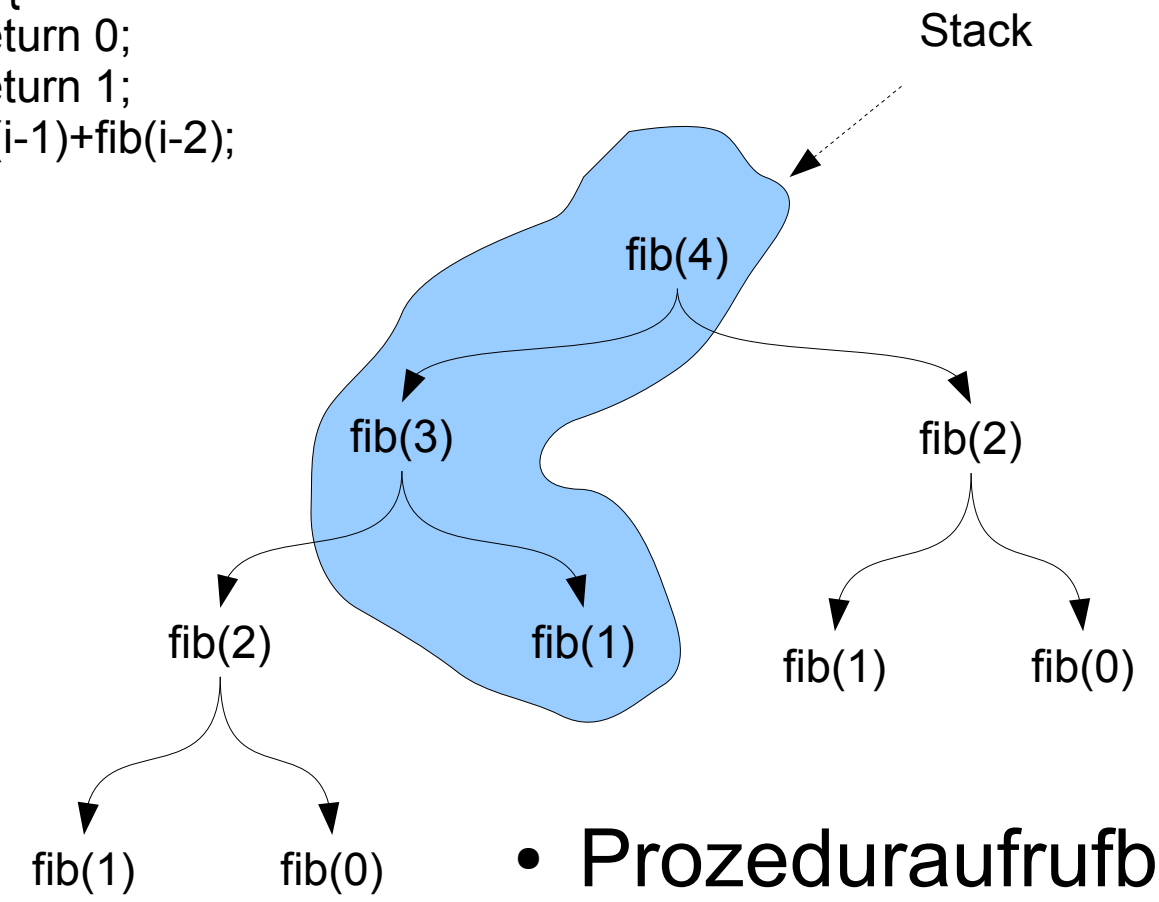
Seine Baumknoten sind mit Prozedurinkarnationen markiert, so dass jede Inkarnation PI genau die Inkarnationen als Kinder hat, die von PI aus erzeugt wurden und zwar in der Reihenfolge des Ablaufs.

Prozeduraufrufsbaum

```
int fib(int i) {
  if(i==0) return 0;
  if(i==1) return 1;
  return fib(i-1)+fib(i-2);
}
```



```
int fib(int i) {
  if(i==0) return 0;
  if(i==1) return 1;
  return fib(i-1)+fib(i-2);
}
```



- Prozeduraufrufbaum = Geschichte der Aufrufe
- Stack = Datenstruktur, gegenwärtige Aufrufe

Visualisierung - Prozedurbaum

```
int depth = 0; // Groesse des Stacks

int fib(int i) {
  depth++; indent(4*depth); print("begin fib("+i+")");

  int result;
  if(i==0) result = 0;
  else if(i==1) result = 1;
  else result = fib(i-1)+fib(i-2);

  indent(4*depth); print("end fib("+i+")"); --depth;
  return result;
}

print(fib(4));
```

Visualisierung - Prozedurbaum (2)

```
begin fib(4)
    begin fib(3)
        begin fib(2)
            begin fib(1)
            end fib(1)
            begin fib(0)
            end fib(0)
        end fib(2)
        begin fib(1)
        end fib(1)
    end fib(3)
    begin fib(2)
        begin fib(1)
        end fib(1)
        begin fib(0)
        end fib(0)
    end fib(2)
end fib(4)
```

Visualisierung - Prozedurbaum (3)

- Buchführung über Tiefe durch Inkrement/Dekrement am Anf./Ende
- nützliche Technik für Tracing
- call/return sind geklammert
- vgl. kontextfreie Grammatik / Stack / Klammerung

Bemerkung:

Die Lebensdauern von Inkarnationen der gleichen Prozedur können sich überlappen.

Deshalb muss es ggf. mehrere Kopien/Instanzen der gleichen lokalen Variablen geben.

Unterabschnitt 4.2.5

Listenalgorithmen in Java

Listentyp

```
public class List {
    int head;
    List tail;

    // NB: Can write procedures inside the data type definition.

    static List cons(int head,List tail) {
        List result = new List();
        result.head = head;
        result.tail = tail;
        return result;
    }
};
```

Listentyp (2)

```
static int length(List l) {
    int n = 0;
    while(l!=null) {
        n++;
        l = l.tail;
    }
    return n;
}
```

Listentyp (3)

```
static List list() {
    return null;
}
static List list(int x) {
    return cons(x,null);
}
static List list(int x,int y) {
    return cons(x,cons(y,null));
}
static List list(int x,int y,int z) {
    return cons(x,cons(y,cons(z,null)));
}
```

Beachte: Overloading

Listentyp (4)

```
static void print(List l) {
    bool first = true;
    System.out.print("(");
    while(l!=null) {
        if(first) first = false; else System.out.print(" ");
        System.out.print(""+l.head);
        l = l.tail;
    }
    System.out.println(")");
}
```

Beachte: ""+value, `first` flag

Listentyp (5)

```
static List prepend(List l,int x) {
    return cons(x,l);
}

static List last(List l) {
    while(l.tail!=null) l = l.tail;
    return l;
}
```

Listentyp (6)

```
static List append1(List l,int x) {
    if(l==null) return cons(x,null);
    last(l).tail = cons(x,null);
    return l;
}

List l = List.list(1,2,3);
ListAlg.append1(l,99);      // FALSCH
l = ListAlg.append1(l,99); // RICHTIG
```

Listentyp (7)

```
    static List conc(List l,List l2) {
        if(l==null) return l2;
        last(l).tail = l2;
        return l;
    }
```

```
bsh % l = List.list(1,2,3);
bsh % m = List.list(4,5,6);
bsh % r = ListAlg.conc(l,m);
bsh % List.print(r);
(1 2 3 4 5 6)
bsh % List.print(l); // BEACHTEN
(1 2 3 4 5 6)
bsh %
```

Listentyp (8)

```
static List insert_sorted(List initial,int x) {
    List l = initial;

    if(l==null) return List.cons(x,null);

    List last = null;
    while(l!=null && x>l.head) {
        last = l;
        l = l.tail;
    }

    if(last==null) return cons(x,l);

    last.tail = cons(x,l);
    return initial;
}
```

Listentyp (9)

```
static boolean sorted(List l) {
    if(l==null) return true;
    while(l.tail!=null) {
        if(l.head>l.tail.head) return false;
    }
    return true;
}

static boolean member(int x,List l) {
    while(l!=null) {
        if(l.head==x) return true;
    }
    return false;
}
```

Listentyp (10)

```
static List insert_sorted1(List initial,int x) {
    if(!sorted(initial)) error("not sorted");
    int n = length(initial);

    List result = insert_sorted(initial,x);

    if(length(result)!=n+1) error("wrong length of result");
    if(!sorted(result)) error("result not sorted");
    if(!member(x,result)) error("element not found in result");
    return result;
}
```

Listentyp (11)

```
static List insert_sorted1(List initial,int x) {
    assert sorted(initial);
    int n = length(initial);

    List result = insert_sorted(initial,x);

    assert length(result)==n+1;
    assert sorted(result);
    assert member(x,result);

    return result;
}
```

Listentyp (12)

Assertion (Zusicherung):

- `assert` hilft beim Testen
- einige Tests dauren lange und sollten nicht im Produktionsmodus verwendet werden
- `assert` kann ein/ausgeschaltet werden und sollte keine Seiteneffekte haben
- `assert` dokumentiert erwartete Eigenschaften des Programms
- `assert` hilft, Fehler bei Veränderungen des Programms sofort zu identifizieren
- `assert` am Anfang einer Prozedur: ***precondition***
- `assert` am Ende einer Prozedur: ***postcondition***
- `assert` in einer Schleife: i.A. ***loop invariant***

Listentyp (13)

```
delete [] x = []
delete (h:t) x | h==x = delete t x
delete (h:t) x = h : delete t x
```

```
static List delete(List l,int x) {
    if(l==null) return null;
    if(l.head==x) return delete(l.tail,x);
    return cons(l.head,delete(l.tail,x));
}
```

Listentyp (14)

```
static List destructive_delete(List l,int x) {
    if(l==null) return null;
    while(l!=null && l.head==x) {
        l = l.tail;
    }
    if(l==null) return null;
    List start = l;
    while(l.tail!=null) {
        if(l.tail.head==x) {
            l.tail = l.tail.tail;
        } else {
            l = l.tail;
        }
    }
    return start;
}
```

Listentyp (15)

Verlinkte Listen

- “traditionelle” Datenstruktur
- destruktive und nicht-destruktive Funktionen
- nicht-destruktiv wie in Haskell (und ähnlich einfach)
- destruktiv
  - effizienter (warum?)
  - komplexer
  - müssen Ergebnis zuweisen
  - Zuweisung vergessen = manchmal (!) falsch
  - unerwartete Veränderung von Werten von anderen Variablen
- i.A. besser: Felder mit “exponential doubling”

Unterabschnitt 4.2.6

Destruktive Bäume

Bemerkung:

```
class Node {
    int mark;
    Node left;
    Node right;

    static Node make(int x,Node left,Node right) {
        Node result = new Node();
        result.mark = x;
        result.left = left;
        result.right = right;
        return result;
    }
}
```

Bemerkung: (2)

```
static Node insert(Node t,int x) {
    if(t==null)
        return make(x,null,null);
    if(t.mark==x)
        return make(t.mark,t.left,t.right);
    if(t.mark>x)
        return make(t.mark,insert(t.left,x),t.right);
    if(t.mark<x)
        return make(t.mark,t.left,insert(t.right,x));
}
```

Bemerkung: (3)

```
static Node destructive_insert(Node t,int x) {
    if(t==null)
        return make(x,null,null);
    if(t.mark==x)
        return t;
    if(t.mark>x)
        t.left = insert(t.left,x);
    else if(t.mark<x)
        t.right = insert(t.right,x);
    return t;
}
```

Unterabschnitt 4.2.7

Manuelle Speicherverwaltung

Bemerkung:

Manuelle Speicherverwaltung ist *notwending* in Sprachen, wie C.

In Java ist manuelle Speicherverwaltung (mittels *Pools*) *optional* und kann zur Beschleunigung von Programmen verwendet werden.

Manuelle Speicherverwaltung mit Listen

```
public class List {
    int head;
    List tail;
    static List cons(int head,List tail) ...
    static void free(List l) ...
    static void used(String msg) ...
}
```

Manuelle Speicherverwaltung mit Listen (2)

```
bsh % List.used();
in use: 0
bsh % l = List.cons(3,null);
bsh % List.used();
in use: 1
bsh % List.free(l);
bsh % List.used();
in use: 0
bsh %
```

Manuelle Speicherverwaltung mit Listen (3)

```
static List destructive_delete(List l,int x) {
    if(l==null) return null;
    while(l!=null && l.head==x) {
        List temp = l;
        l = l.tail;
        List.free(temp);
    }
    if(l==null) return null;
    List start = l;
    while(l.tail!=null) {
        if(l.tail.head==x) {
            List temp = l.tail;
            l.tail = l.tail.tail;
            List.free(temp);
        } else {
            l = l.tail;
        }
    }
    return start;
}
```

Manuelle Speicherverwaltung mit Listen (4)

Memory Leak

```
bsh % List.used();
in use: 0
bsh % l = List.cons(3,null);
bsh % List.used();
in use: 1
bsh % m = List.cons(4,null);
bsh % List.used();
in use: 2
bsh % List.free(m);
bsh % List.used();
in use: 1
bsh % l= List.cons(5,null);
bsh % List.used();
in use: 2
bsh % List.free(l);
bsh % List.used();
in use: 1
bsh %
```

Manuelle Speicherverwaltung mit Listen (5)

Dangling Reference (baumelnder Zeiger)

```
bsh % l = List.cons(3,null);
bsh % List.free(l);
bsh % m = List.cons(4,null);
bsh % List.print(m);
(4)
bsh % l.head = 99;
bsh % List.print(m);
(99)
bsh %
```

Listen im Speicher

```
public class PC {
    int[] memory;
}
public class List {
    int head;
    int tail;
    static int head(int l) {
      return PC.memory[l];
    }
    static int tail(int l) {
      return PC.memory[l+1];
    }
    static int set_head(int l,int x) {
      PC.memory[l] = x;
    }
    static int set_tail(int l,int l2) {
      PC.memory[l+1] = l2;
    }
    ...
};
```

Listen im Speicher (2)

```
static int cons(int head,int tail) {
  int l = ... find free pair ...;
  PC.memory[l] = head;
  PC.memory[l+1] = tail;
}
static void free(List l) {
  ...
}
```

Listen im Speicher (3)

Fehlerchecking (oft größtenteils in Hardware):

```
public class List {
    int head;
    int tail;
    static int NULL = -1;
    static int head(int l) {
      assert l!=NULL && l%2==0 && l<PC.memory.length && l>=0;
      return PC.memory[l];
    }
    ...
};
```

Listen im Speicher (4)

Grundidee:

Freie Elemente können auch als Liste repräsentiert werden (s.o.).

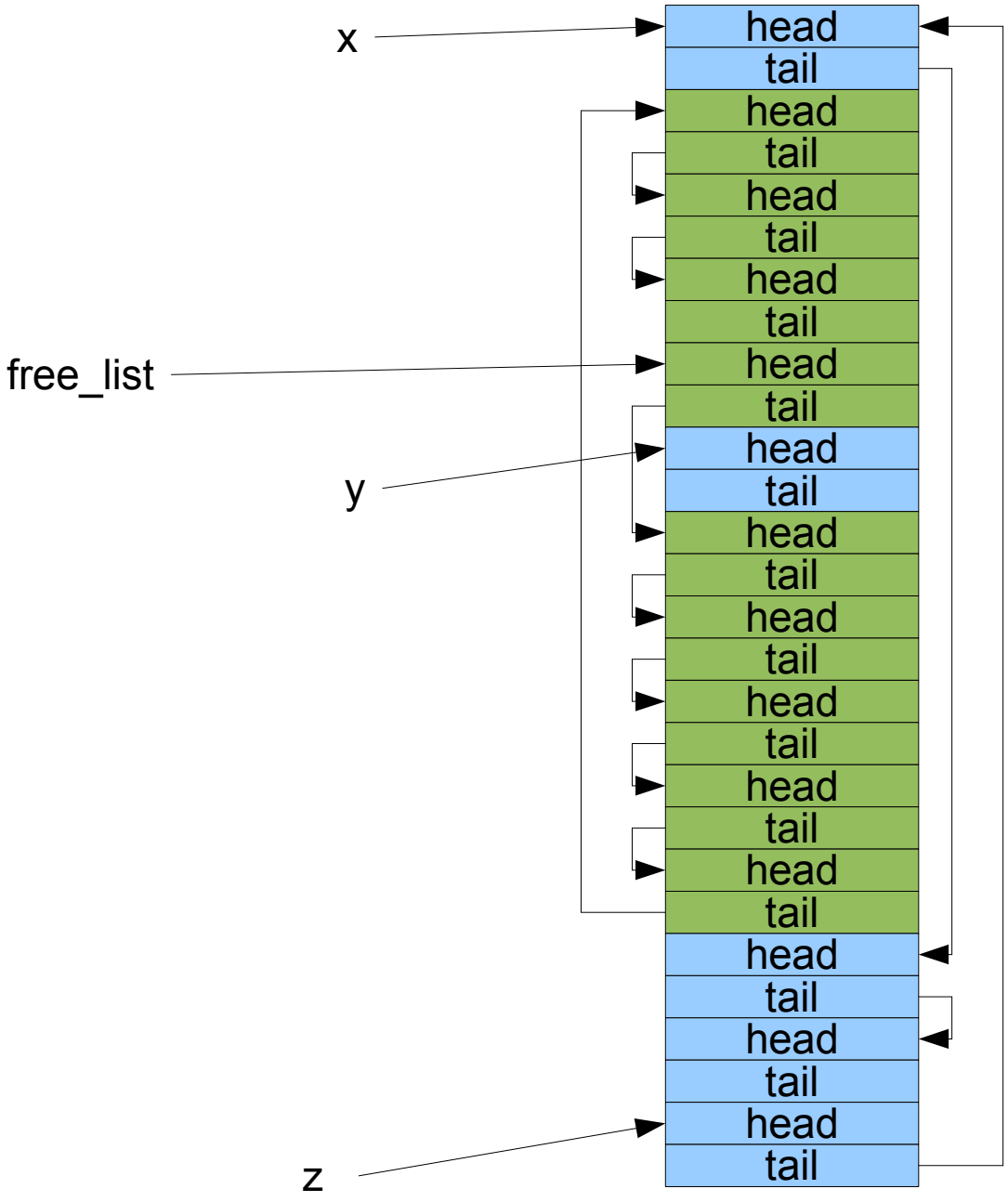
Erzeuge eine Liste aller Elemente am Anfang, dann gib daraus freie Elemente zurück.

Listen im Speicher (5)

```
static int free_list;

static void init_free_list() {
  free_list = 0;
  int i;
  for(i=0;i<PC.memory.length-1;i+=2) {
    PC.memory[i+1] = i+2;
  }
  PC.memory[i+1] = NULL;
}
```

Listen im Speicher (6)

```
static int free_list;

static int cons(int head,int tail) {
  assert free_list!=NULL;
  int result = free_list;
  free_list = tail(free_list);
  PC.memory[result] = head;
  PC.memory[result+1] = tail;
}

static void free(int l) {
  set_head(l,-1);
  set_tail(l,free_list);
  free_list = l;
}
```


x
free_list
y
z
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail
head
tail

Unterabschnitt 4.2.8

Felder als Listen

Felder als “Listen”

```
public class Flist {
    int[] data;
    int fill;

    static Flist make(int n) {
        Flist l = new Flist();
        l.data = new int[n];
        l.fill = 0;
    }
    static void push(Flist l,int x) {
        l.data[fill++] = x;
    }
    static int pop(Flist l) {
        return l.data[--fill];
    }
    static int at(Flist l,int i) {
        return l.data[i];
    }
}
```

Felder als “Listen” (2)

```
public class Flist {
    int[] data;
    int fill;

    static Flist make(int n) {
        assert n>0;
        Flist l = new Flist();
        l.data = new int[n];
        l.fill = 0;
    }
    static void push(Flist l,int x) {
        assert fill<data.length;
        data[fill++] = x;
    }
    static int pop() {
        assert fill>0;
        return data[--fill];
    }
    static int at(int i) {
        assert i>=0 && i<data.length;
        return data[i];
```

Felder als “Listen” (3)

```
        }
}
```

Felder als “Listen” (4)

exponentielles Wachstum

```
public class Flist {
    int[] data;
    int fill;

    static void push(Flist l,int x) {
        if(l.fill==l.data.length) {
            int ndata = new int[2*l.data.length];
            for(int i=0;i<fill;i++) ndata[i] = l.data[i];
            l.data = ndata;
        }
        assert fill<data.length;
        data[fill++] = x;
    }
    ...
}
```

Felder als “Listen” (5)

Aufwandsabschätzung: Hinzufügen von 1024 Elementen

verlinkte Listen: rufe `cons` 1024 mal auf, 1024 Zuweisungen

Stacks: rufe `new int` 9 mal auf mit Größen 1-1024, kopiere $1 + 2 + 4 + 8 + ... + 512$ mal Elemente in `push`, dann weise insgesamt 1024 mal zu.

Element 1025 führt zu Vergrößerung auf 2048; diese große Arbeit wird dann aber *amortisiert*, und wenn wir mit n Elementen arbeiten ist der Aufwand dafür immer höchstens $2n$

Felder als “Listen” (6)

Speicheraufand

verlinkte Listen: 1024 Worte für heads, 1024 Worte für tails (plus 3×1024 für Java!)

Stacks: aktiv ein Feld von 1024 Elementen = 1024 Worte (plus 3 für Java)

Element 1024 führt zum Verbrauch von 2048 Worten, aber immer noch nicht höher als verlinkte Listen

Felder als “Listen” (7)

Operation	verlinkt	feldbasiert
erzeugen von List mit n Elementen	$O(n)$	$O(n)$
Bestimmen der Länge	$O(n)$	$O(1)$
voranstellen / prepend	$O(1)$	$O(n)$
anhängen / append	$O(n)$	$O(1)$
Zugriff auf Element in der Mitte	$O(n)$	$O(1)$
Einfügen in der Mitte	$O(n)$	$O(n)$
Löschen aus der Mitte	$O(n)$	$O(n)$

- $O(1)$ = Anzahl von Schritten unabhängig von Listenlänge
- $O(n)$ = Anzahl von Schritten i.A. proportional zu Listenlänge
- $O(n^2)$ = Anzahl von Schritten i.A. proportional zum Quadrat der Listenlänge

Dies wird später noch vertieft.

Felder als “Listen” (8)

“Listen” in vielen modernen Programmiersprachen sind feldbasiert, nicht verlinkt.

Effizienz wird durch exponentielles Wachstum der Feldgröße bei Bedarf erreicht.

Aufwand ist i.A. gleich oder besser als Listen.

Achtung: bei verlinkten Listen ist voranstellen einfacher, bei feldbasierten Listen das Anhängen.

Felder als “Listen” (9)

Stimmt die Analyse noch für dies?

```
if(fill==data.length) {
    int ndata = new int[data.length+data.length/5];
    for(int i=0;i<fill;i++) ndata[i] = data[i];
    l.data = ndata;
}
```

Felder als “Listen” (10)

Stimmt die Analyse noch für dies?

```
if(fill==data.length) {
    int ndata = new int[data.length + 1000];
    for(int i=0;i<fill;i++) ndata[i] = data[i];
    l.data = ndata;
}
```

Unterabschnitt 4.2.9

Felder im Speicher

Felder im Speicher

```
public class PC {
  static int[] memory;
}
```

Felder im Speicher (2)

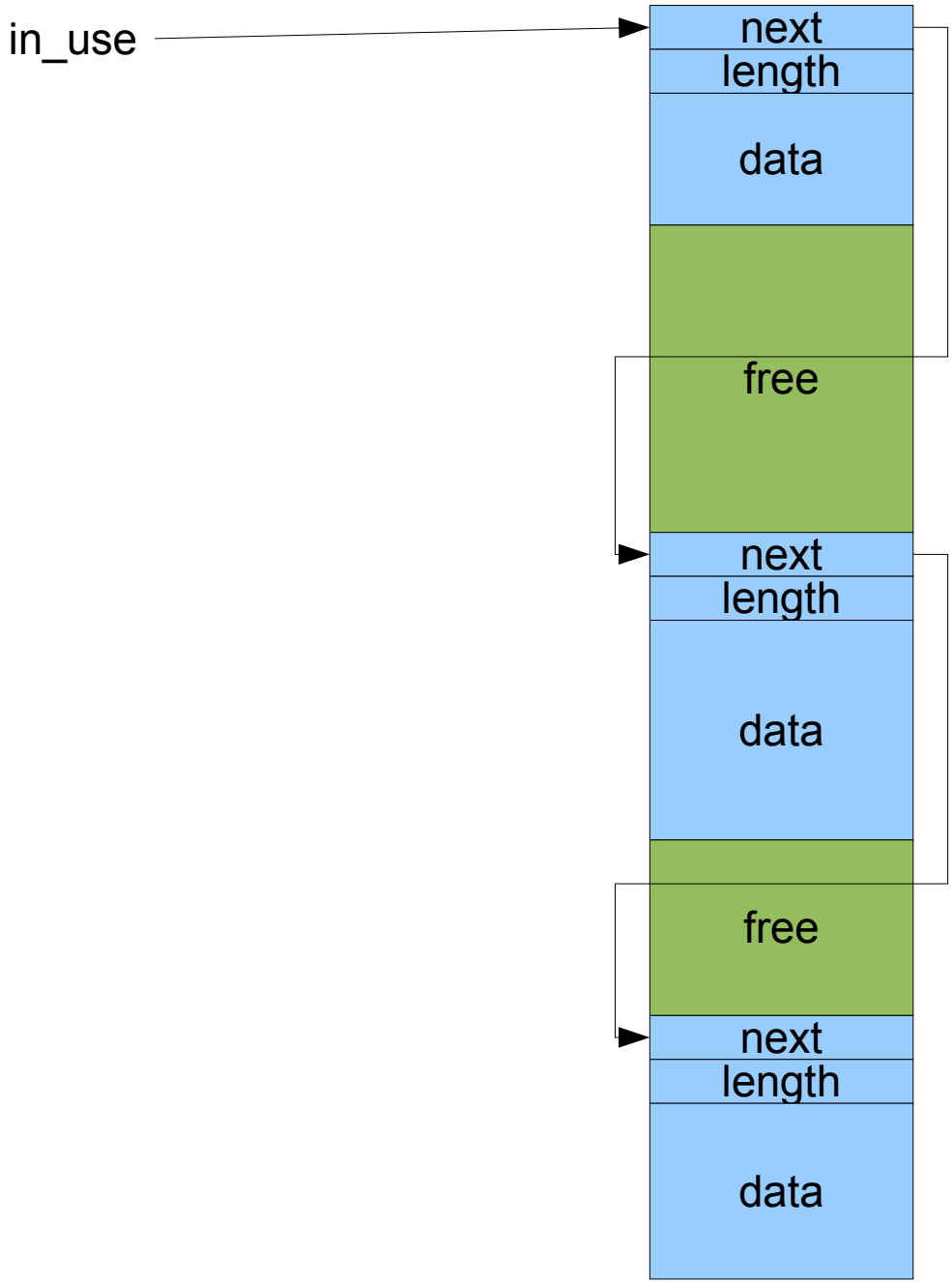
```
public class MyArray {
  int start;
  int length;

  static int get(MyArray a,int index) {
    assert index>=0 && index<a.length;
    return PC.memory[a.start+index];
  }
  static void put(MyArray a,int index,int value) {
    assert index>=0 && index<a.length;
    PC.memory[a.start+index] = value;
  }
  static MyArray alloc(int n) {
    ... finde unbenutzten Bereich von Laenge n ...
    ... markiere ihn als benutzt ...
    ... gib einen Deskriptor zurueck ...
  }
  void free(MyArray a) {
    ...
  }
}
```

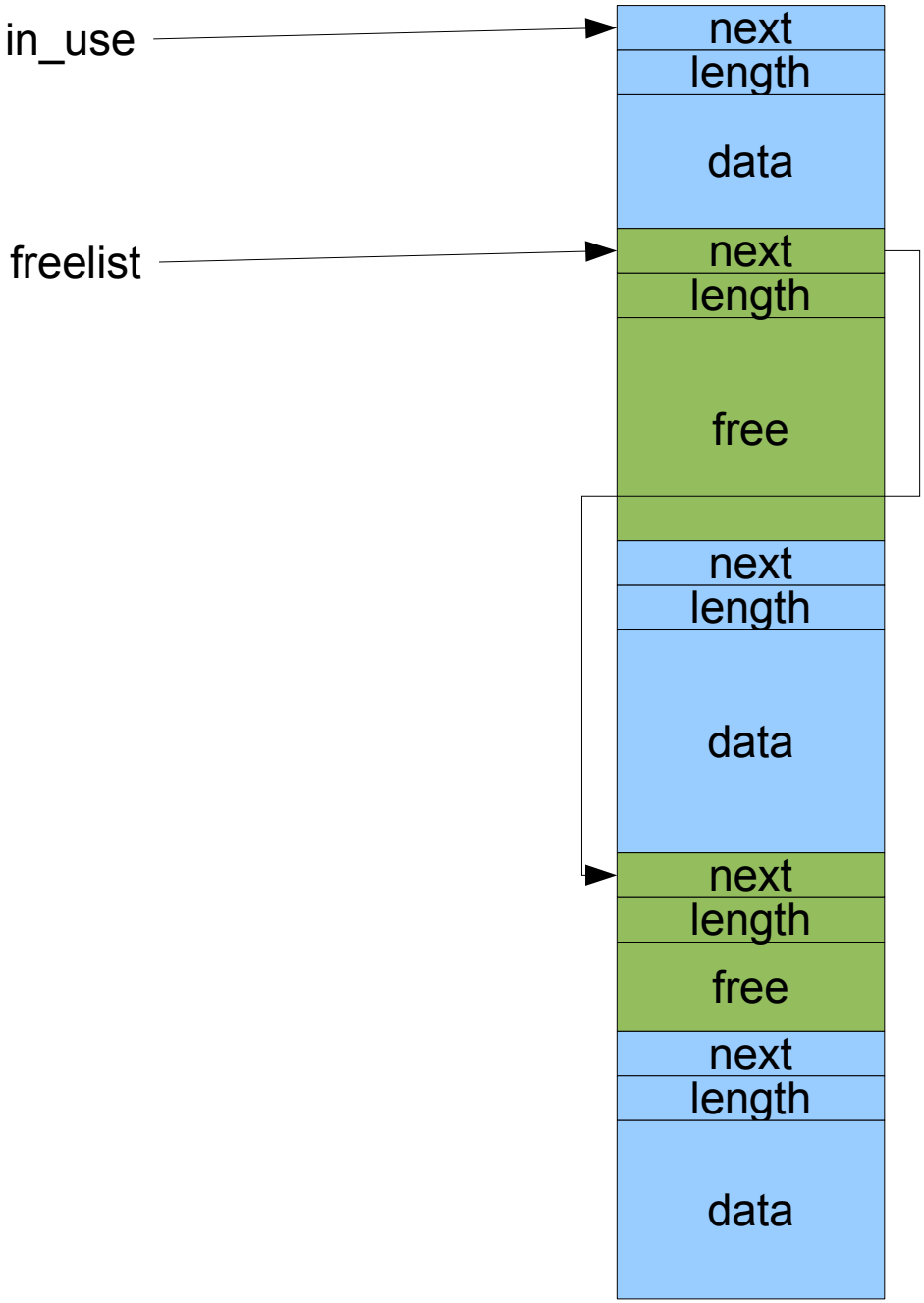
Felder im Speicher (3)

```
public class MyArray {
  static int alloc(int n) {
    ... finde unbenutzten Bereich von Laenge n+2 ...
    ... markiere ihn als benutzt ...
    ... gib den Index des ersten Elements im Speicher zurueck ...
  }
  static void free(int a) {
    ...
  }
  static int get(int a,int index) {
    int length = PC.memory[a-1];
    assert index>=0 && index<length;
    return PC.memory[a+index];
  }
  static void put(int a,int index,int value) {
    int length = PC.memory[a-1];
    assert index>=0 && index<length;
    PC.memory[a+index] = value;
  }
}
```

Felder im Speicher (4)

Im Speicher werden Felder als fortlaufende Bereiche von Worten dargestellt und durch einen ***Zeiger*** repräsentiert.
Ein Zeiger ist die Nummer des Wortes wo die Element des Feldes anfangen.
Zusätzliche Worte werden für Speicherverwaltung benötigt.
Einfache Speicherverwaltung: verlinkte Liste von gebrauchten oder gebrauchten+benutzten Blöcken.
Implementierung? Versuchen Sie sich dran.


in_use
next
length
data
free
next
length
data
free
next
length
data


in_use
freelist
next
length
data
next
length
free
next
length
data
next
length
free
next
length
data

Unterabschnitt 4.2.10

Garbage Collection

Mini-Facebook

We define a very simple “Facebook” application. There are members and there are pages. Each member has a name and a number of pages. Pages have a title and a list of related pages.

```
class Member {
    String name;
    Page[] pages;
}

class Page {
    String title;
    Page[] related;
    boolean marked;
}

class Facebook {
    Member[] members;
    Page[] pages;
}
```

Mini-Facebook (2)

```
print("adding members");

Facebook fb = new Facebook();
fb.members = new Member[100];
for(int i=0;i<fb.members.length;i++) {
    fb.members[i] = new Member();
    fb.members[i].name = "Member #"+i;
    fb.members[i].pages = new Page[100];

}

... same for pages ...
```

Mini-Facebook (3)

Users can add pages to their member profile.

```
boolean add_page(Member member,Page page) {
    if(member==null || page==null) return false;
    for(int i=0;i<member.pages.length;i++) {
        if(member.pages[i]!=null) continue;
        member.pages[i] = page;
        return true;
    }
    return false;
}
```

Mini-Facebook (4)

Users can add pages to other pages.

```
boolean add_related(Page page,Page related) {
    if(page==null || page==null || page==related) return false;
    for(int i=0;i<page.related.length;i++) {
        if(page.related[i]!=null) continue;
        page.related[i] = related;
        return true;
    }
    return false;
}
```

Dies erzeugt ein ***Geflecht***

Mini-Facebook (5)

Now we simulate user activity. These kinds of simulations are quite common in computer science and let us test systems and measure their behavior.

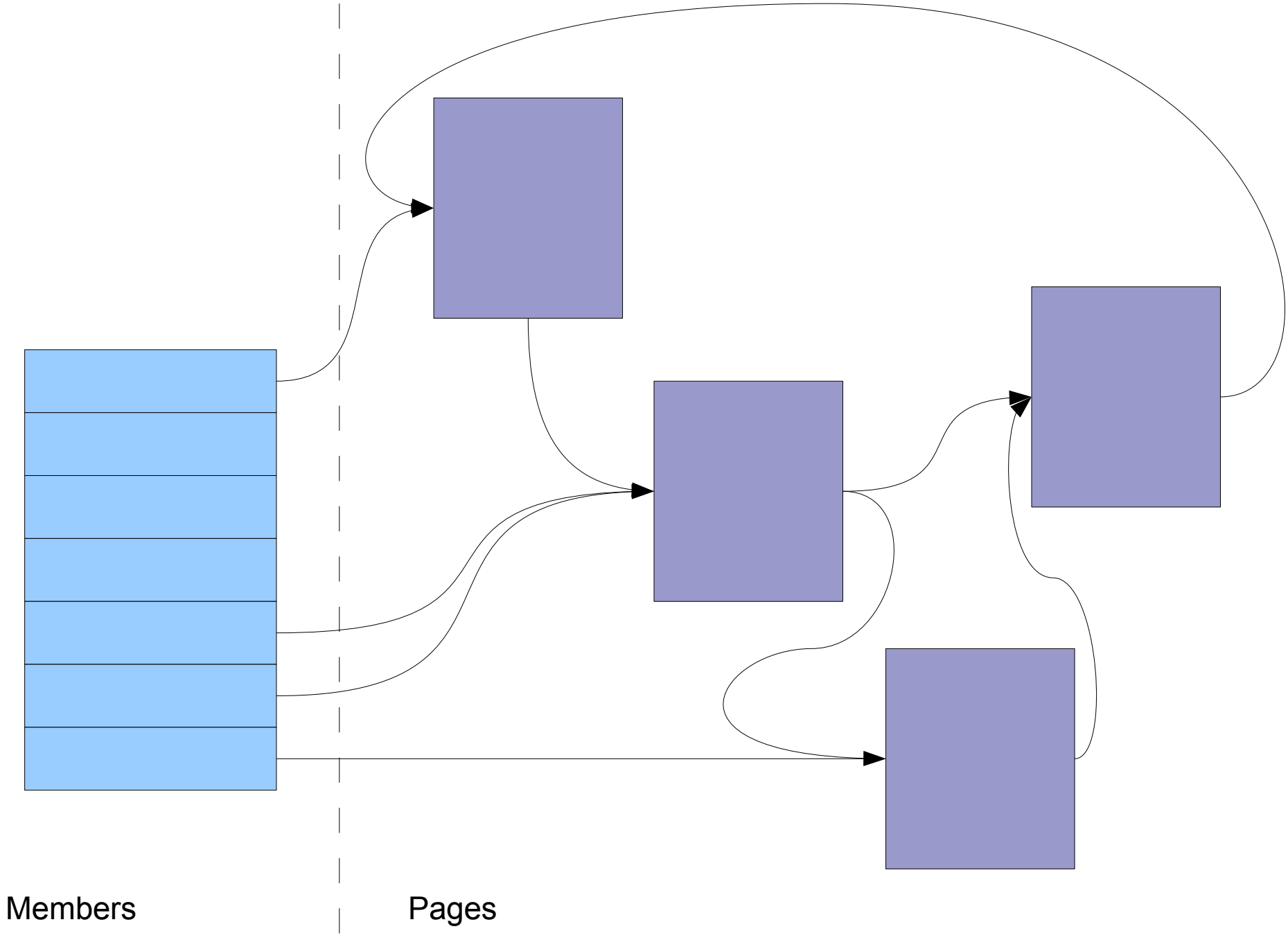
We start off by picking a random member and a random page and have the user add that page to his profile.

```
print("adding pages to member profiles");

for(int i=0;i<fb.pages.length;i++) {
    for(int j=0;j<3;j++) {
        Page page = fb.pages[random.nextInt(fb.pages.length)];
        Page related = fb.pages[random.nextInt(fb.pages.length)];
        add_related(page,related);
    }
}

... same for members ...
```

Mini-Facebook (6)

After all this user activity, many pages are either directly on some member's list of pages, or they are at least related to some other page, even if they are not directly on someone's list of pages.

However, some pages have become "unreachable": that is, they are neither linked to directly from a member profile, nor are they linked to from another page. We want to find these unreachable pages so that we can delete them.


Members
Pages

Mini-Facebook

Markieren aller Seiten von Mitgliedern:

```
void mark(Member member) {
    if(member==null) return;

    // Marking a member is simple: we just step through each
    // page in his profile and mark that page.

    for(int i=0;i<member.pages.length;i++)
        mark(member.pages[i]);
}
```

Mini-Facebook (2)

Markieren aller Seiten, auf die sich andere Seiten beziehen.

```
void mark(Page page) {
    if(page==null || page.marked) return;

    // First, we mark the page as having been visited.  We do
    // this before we mark related pages so that we don't recurse
    // on the same page twice.

    page.marked = true;

    // Now we step through the list of related pages and
    // mark all of those as well.

    for(int i=0;i<page.related.length;i++)
        mark(page.related[i]); // REKURSIV
}
```

Mini-Facebook (3)

```
void remove_unreachable_pages() {

    // First, we clear all the marks on all the known pages.

    print("clearing marks");

    for(int i=0;i<fb.pages.length;i++)
        fb.pages[i].marked = false;

    // Next, we mark all the pages that are reachable from a member
    // profile.

    print("marking pages on member profiles");

    for(int i=0;i<fb.members.length;i++)
        mark(fb.members[i]);
```

Mini-Facebook (4)

```
    // After the recursive marking step, any page that has not been
    // marked is neither listed in a member profile, nor listed as
    // related to any page that is reachable from a member profile.
    // Those are the pages we can delete.

    print("deleting unreachable pages");

    ndeleted = 0;
    for(int i=0;i<fb.pages.length;i++) {
        if(fb.pages[i].marked) continue;
        print("deleting: "+fb.pages[i].title);
        // We delete the page simply by setting its corresponding
        // entry to null.  Of course, the Java storage manager
        // eventually cleans up the memory associated with that page.
        fb.pages[i] = null;
        ndeleted++;
    }

    print("deleted "+ndeleted+" unreachable pages");
}
```

Mini-Facebook (5)

Garbage Collection

- ***Roots*** (Wurzeln) – Objekte, die namentlich referenziert werden können
- ***Object Graph*** (Geflecht) – Objekte, die sich gegenseitig referenzieren
- ***Reachability*** (Errichbarkeit) – Objekte, zu denen wir von den Wurzeln durch Referenzen navigieren können.
- ***Mark Phase*** – Markierung aller Objekte, die von den Wurzeln erreichbar sind.
- ***Sweep Phase*** – Aufräumen aller Objekte, die nicht markiert sind.

Mini-Facebook (6)

Konzept	Mini-Facebook	Java
Wurzeln	Personen	Variablen
Objekte	Seiten	beliebige Objekte
Mark Phase	rek. Markierung Seiten	rek. Markierung Objekte
Sweep Phase	Entfernung von Seiten	Rückgabe des Speichers

Mini-Facebook (7)

- Garbage Collection in Java ist analog zu anderen Datenverwaltungsproblemen
- Garbage Collection is praktisch: wir brauchen uns keine (?) Gedanken um Speicherverwaltung zu machen
- Garbage Collection ist wichtig, um Speicherverwaltungsfehler zu vermeiden (dangling references, storage leaks)
- Amortisiert über Objekte ist Garbage Collection oft effizienter, als manuelle Speicherverwaltung.

Unterabschnitt 4.2.11

Effizienz

praktische Effizienz

- ***Laufzeit*** – Anzahl der Schritte, die ein Programm braucht
  Messen durch Stoppuhr oder Zählen der Schritte/Aufrufe (auch mittels Instrumentierung).
- ***Speicherbedarf*** – **maximale** Anzahl der Speicherworte, die das Programm benötigt
  Messen durch Instrumentierung der Speicherfunktionen

praktische Effizienz (2)

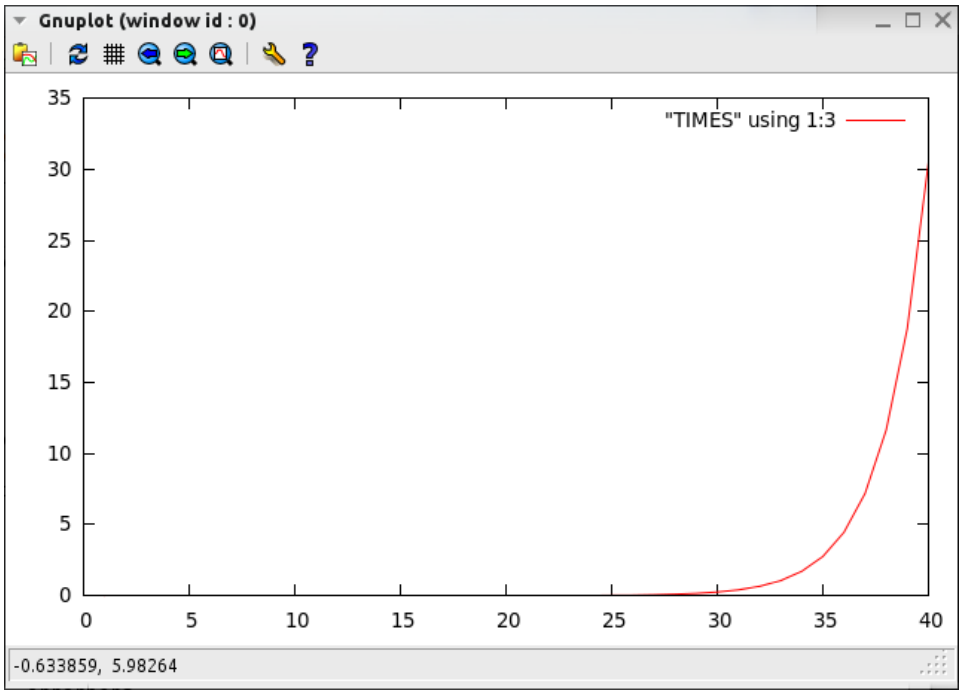
```
$ cat fib.hs
import System (getArgs)

fib 0 = 1
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))

fib_main :: [String] -> Int
fib_main [arg] = fib (read arg)

main = do
  args <- getArgs
  print (fib_main args)
$
```

praktische Effizienz (3)

```
$ seq 40 | while read x; do
      echo $x $(/usr/bin/time -f '%U\n' a.out $x 2>&1);
  done > TIMES
$ tail TIMES
31 2178309 0.40
32 3524578 0.65
33 5702887 1.04
34 9227465 1.70
35 14930352 2.74
36 24157817 4.44
37 39088169 7.18
38 63245986 11.64
39 102334155 18.79
40 165580141 30.45
```

praktische Effizienz (4)

```
$ gnuplot
...
gnuplot> plot "TIMES" using 1:3 with lines
$
```



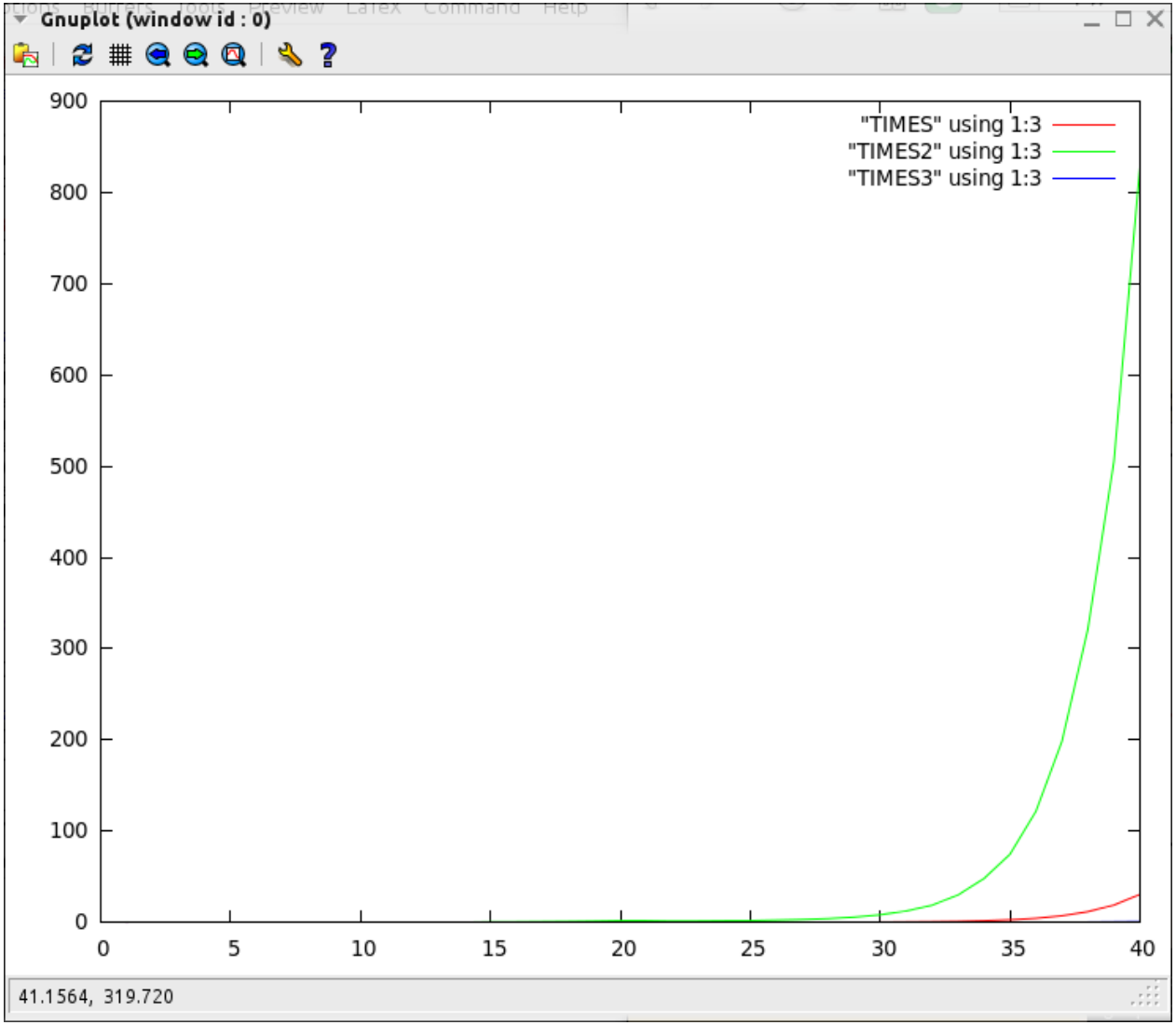
praktische Effizienz (5)

```
$ cat fib.bsh
fib(i) {
    if(i<2) return 1;
    return fib(i-1)+fib(i-2);
}

n = Integer.parseInt(bsh.args[0]);
print(fib(n));
$
```

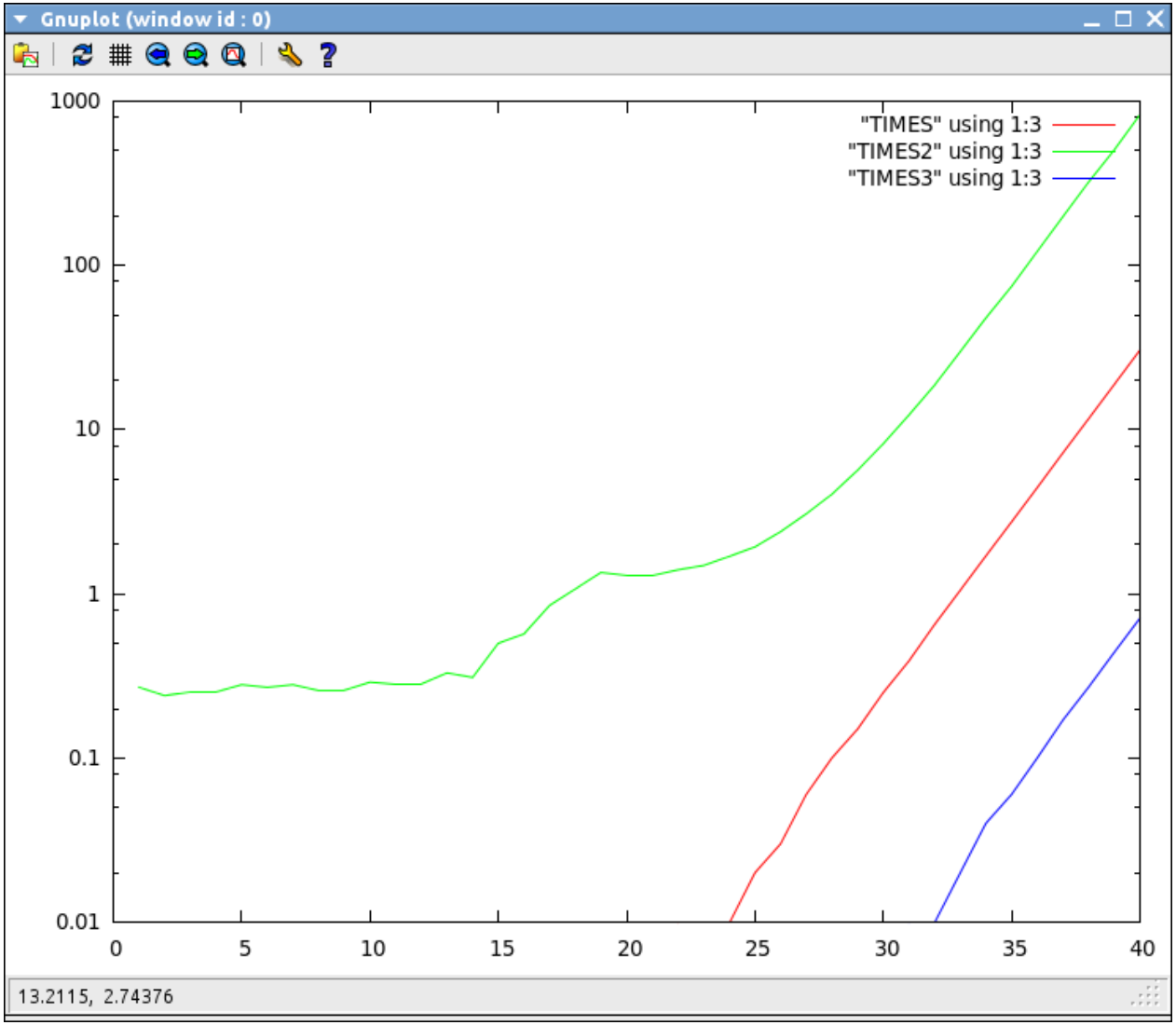
praktische Effizienz (6)

```
$ cat fib.c
#include <stdio.h>

int fib(int n) {
    if(n<2) return 1;
    return fib(n-1)+fib(n-2);
}

int main(int argc,char **argv) {
    int n = atoi(argv[1]);
    printf("%d\n",fib(n));
    return 0;
}
$
```

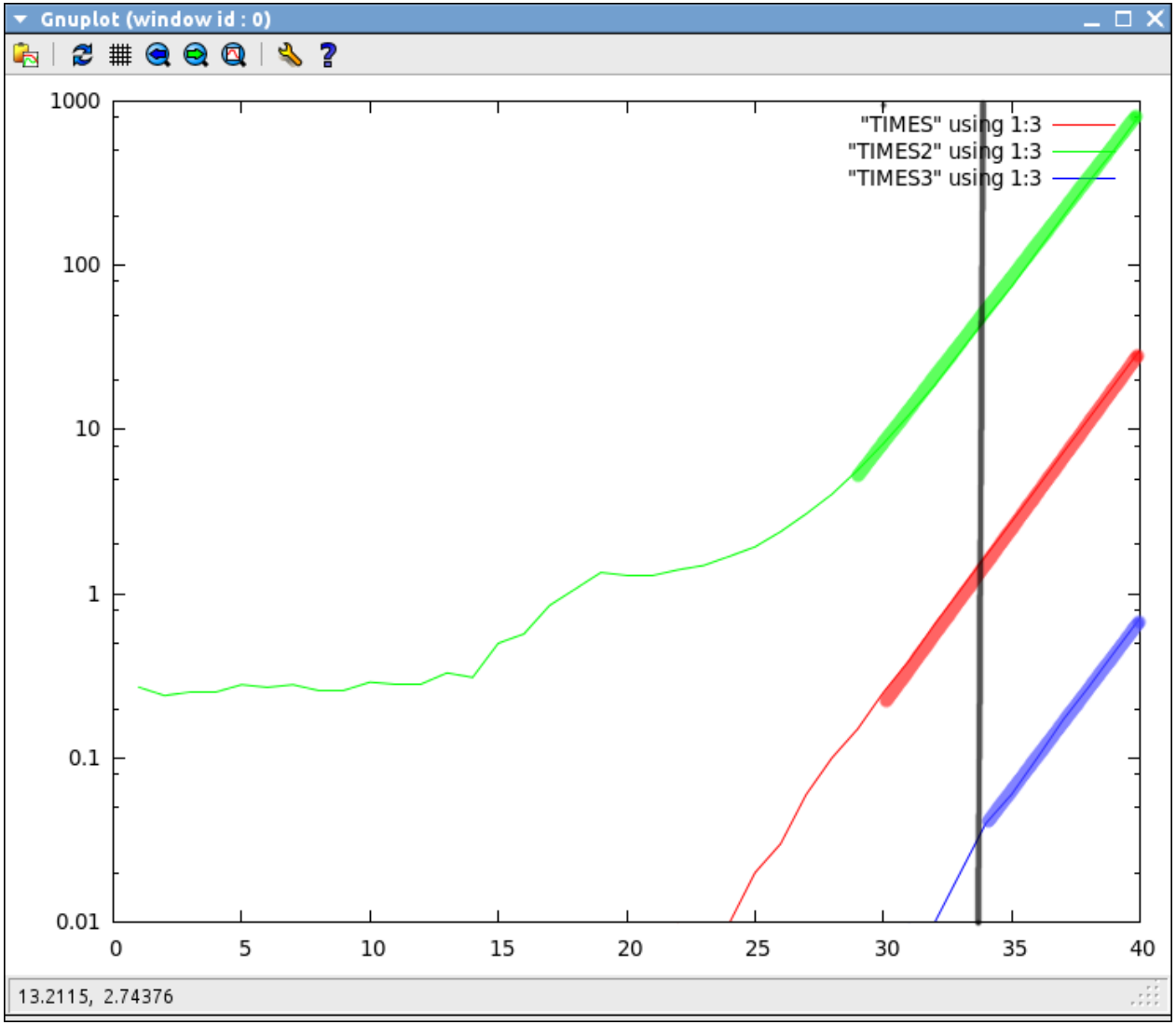
praktische Effizienz (7)



praktische Effizienz (8)



praktische Effizienz (9)



praktische Effizienz (10)

- Laufzeit und Speicherbedarf sind abhängig von Eingabedaten, -werten
- Laufzeit und Speicherbedarf können empirisch bestimmt werden.
- ***Asymptotisch*** wird das Laufzeitverhalten verschiedener Implementierungen desselben Algorithmus ugf. proportional