Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

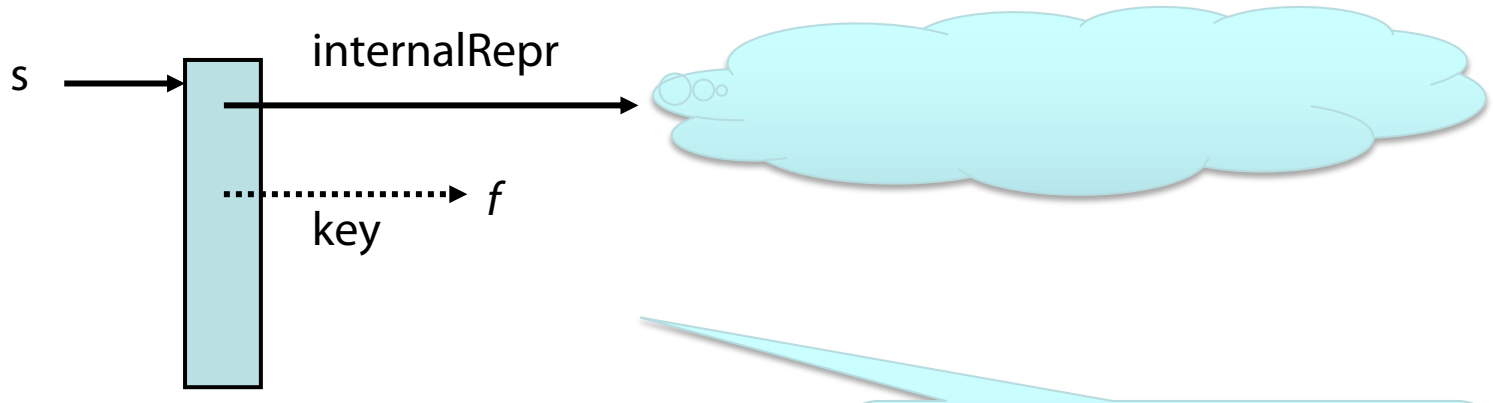
Menge

- Datenstruktur, die Objekte verwaltet, für die Schlüssel (keys) definiert sind:
  - einfache Menge (keine Schlüsselwert-Duplikate enthalten)
  - Multimenge
  - geordnete (Multi-)Menge

```
function identity(x)
  return x
```

- Erzeugung von speziellen Mengen
  - s := {42, 23, 17} // key ist Identitätsfunktion
    - test(23, s) liefert true, test(1024, s) liefert false
  - s := { } // key ist identity
  - s := {…} with key as … // key wird vorgegeben
  - s := <…>:Set with key as … // Alternative Notion
  - s := <…>:OrderedSet … //
  - s := <…>:Multiset … //
  - s := <…>:OrderedMultiset … // Könnte als PQ angesehen werden,
    // hat aber anderes API
- Auch möglich: pq := <…>:PQ with key as …

Mengen



Der Wert von s ist eine **Instanz** des Datentyps Menge

key(s) liefert Funktion *f*

Sei f eine einstellige Funktion

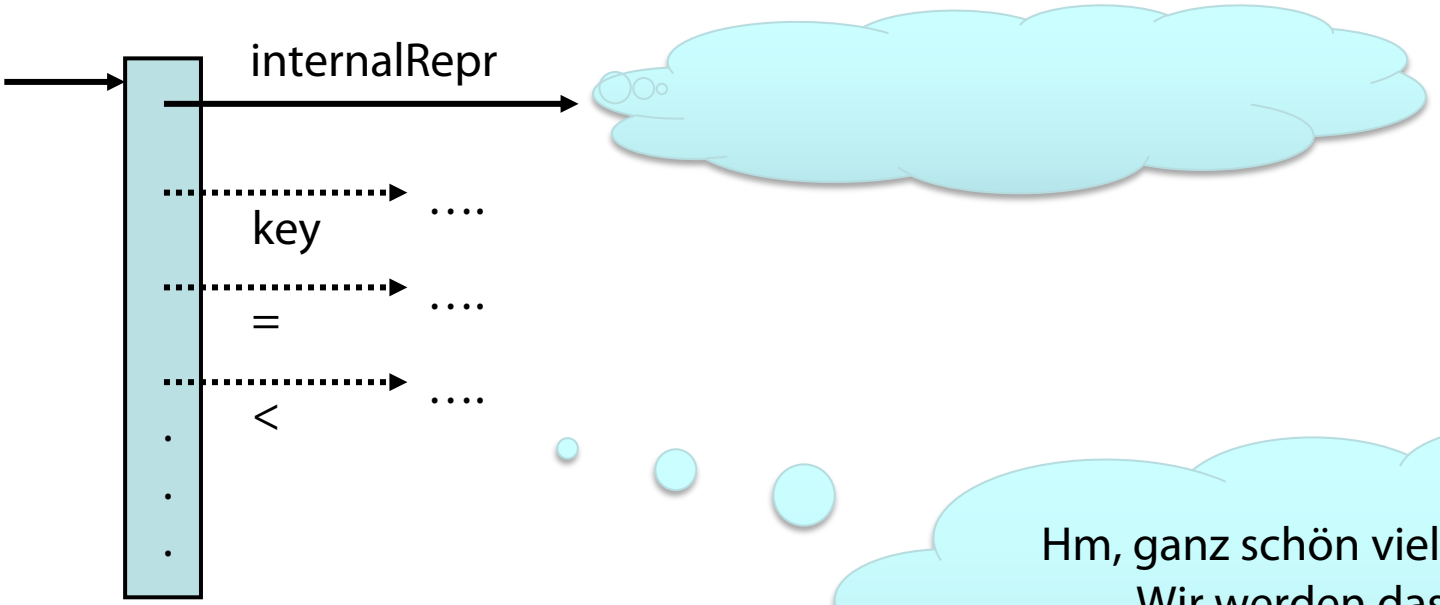
key(s)(x) wendet Funktion f auf x an

Vergleiche mit Zeichenketten

- Zeichenketten (Strings) sind Zeiger auf Arrays mit Buchstaben
- Gilt "Lucky" = "Lucky" ?
  - Nein
  - Die Funktion = vergleicht Zeiger nicht Inhalte

- Problem:
  - test("Lucky", {"Rolly", "Penny", "Lucky"}) wird nie true liefern
- Vergleichsprädikate für Elemente sollten bei Mengen auch angegeben werden können
  - s := {…} with key as … with = as … // Vergleichsprädikat, Standard: =
- Beispiel:
  - **function** stringEqual(str1, str2) … // buchstabenweiser Vergleich
  - s := {"Rolly", "Penny", "Lucky"} with = as stringEqual
  - test("Lucky", s) funktioniert dann
- Wir schreiben =(*set*)(…, …) Beispiel: =(s)(…, …)
- Gleiches für <, ≤, >, ≥ usw.

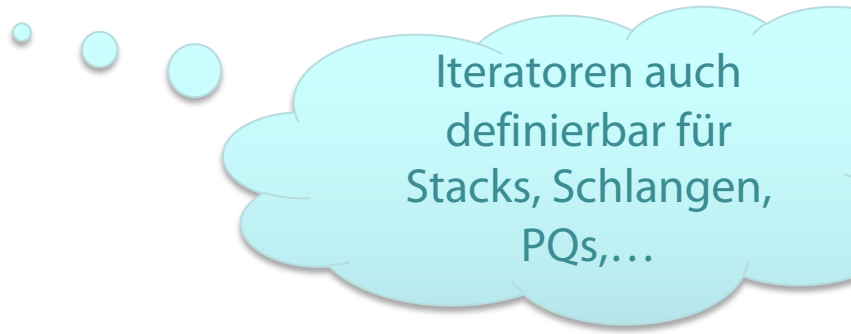
Mengen mit speziellen Eigenschaften



- Gilt denn {1, 2} = {1, 2} ?
  - Nein
- Gilt denn {} = {} ?
  - Nein

Abstrakter Datentyp: Iteratoren

- Wie kann ich über eine Struktur iterieren, dessen interne Repräsentation verborgen ist?
- Iteratoren für einfache Mengen und Multimengen
  - getIterator(s),
  - testNextElement(i), testPreviousElement(i)
  - nextElement(i), previousElement(i)



- Iteratoren für geordnete Mengen
  - getIterator(s, fromKey), getIterator(s, fromKey, toKey)

Iteration über die Elemente von ADTs

- Wir wollen von der genauen Datenstruktur abstrahieren

```
function test(k, s)
  iter := getIterator(s)                  // Erzeuge Iterator mit Zustand
  while testNextElement(iter) do          // Noch ein Element vorhanden?
    x := nextElement(iter)                // Funkt. mit Iteratorzustandsänderung
    if =(s)(k, key(s)(x)) then
      return true
  return false
```

- Kurzschreibweise für die Iteratoranwendung

```
function test(k, s)
  for x ∈ S do
    if =(s)(k, key(s)(x)) then
      return true
  return false
```

Ultrakurzschreibweise

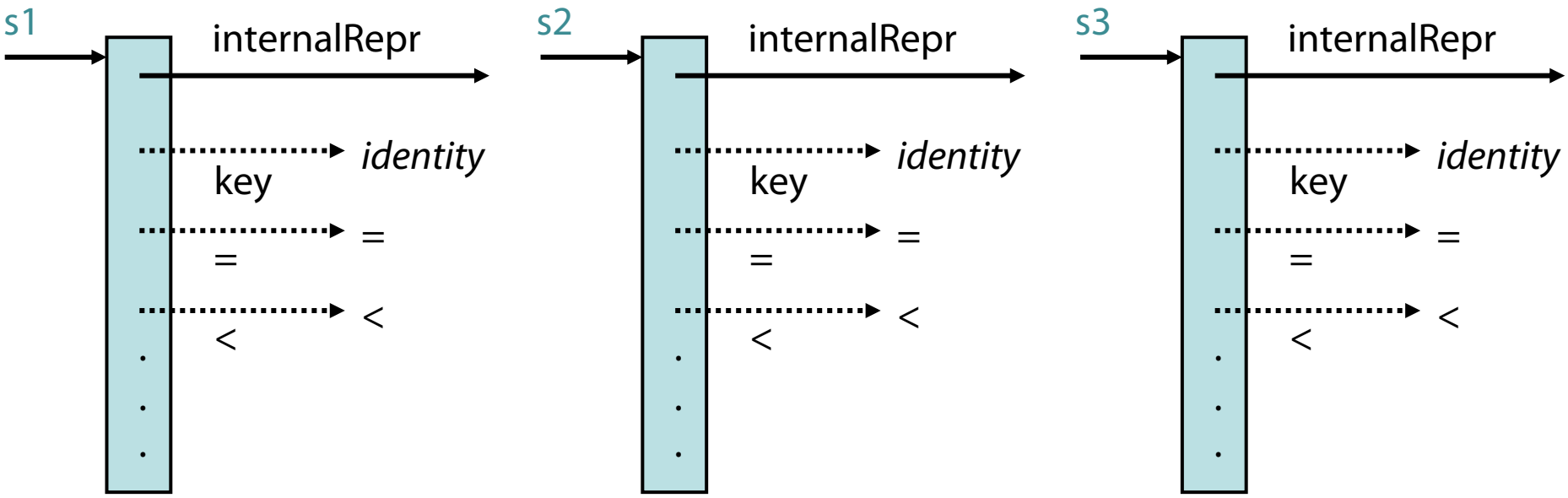
- Doppelpunkt als Kontext

```
function test(k, s:)
  for x ∈ s do
    if k = key(x) then
      return true
  return false
```

```
function test(k, s)
  for x ∈ s do
    if =(s)(k, key(s)(x)) then
      return true
  return false
```

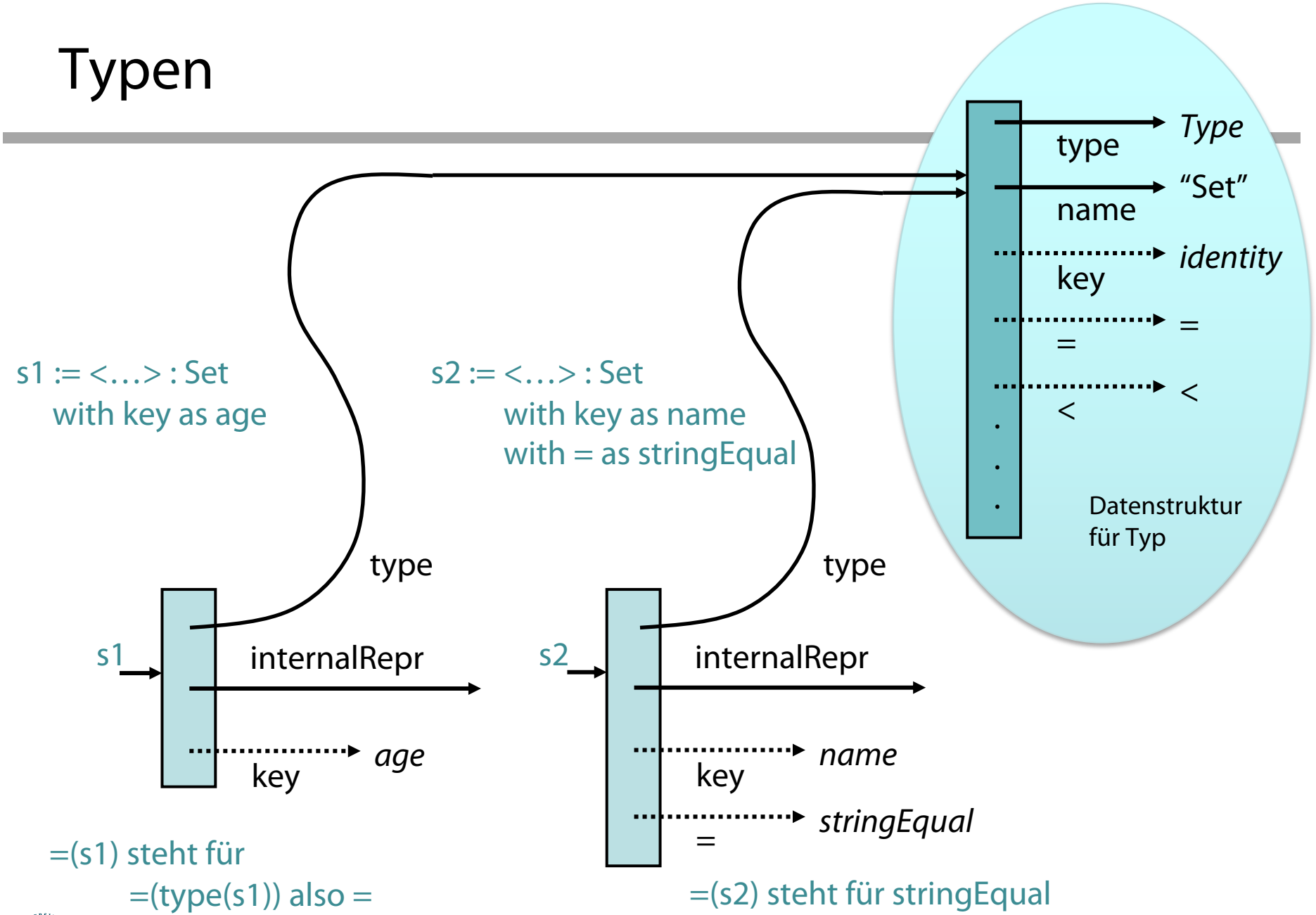
- Wenn im Rumpf spezielle Funktionen (oder Prozeduren) für Kontextvariable (mit : gekennzeichnet) definiert sind, dann werden diese auch verwendet
- Gilt hier für = und key
  - Im Kontext s: steht = für =(s) und key für key(s)

Alles pro Instanz speichern?



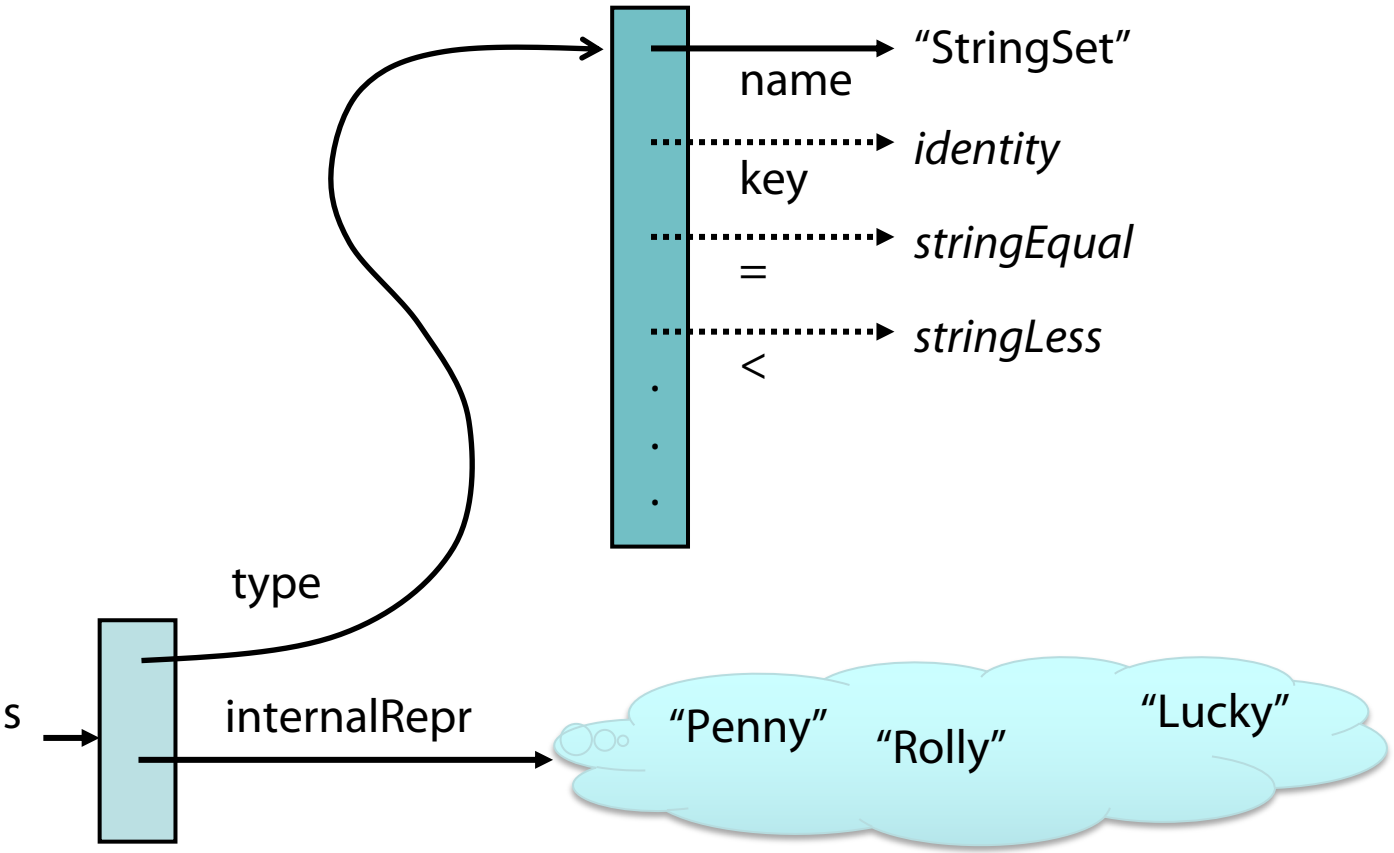
- Zuviel Redundanz?

Typen


type
Type
"Set"
name
identity
key
=
=
<
<
Datenstruktur
für Typ
s1 := <…> : Set
with key as age
s2 := <…> : Set
with key as name
with = as stringEqual
type
s1
internalRepr
key
age
type
s2
internalRepr
name
key
stringEqual
=
=(s1) steht für
=(type(s1)) also =
=(s2) steht für stringEqual

Kontexte und Instanzen

s := <“Rolly”, “Penny”, “Lucky”> : StringSet



Im Kontext s: steht = für =(type(s))

Typdefinition und Methoden

type Set ( ) ( )

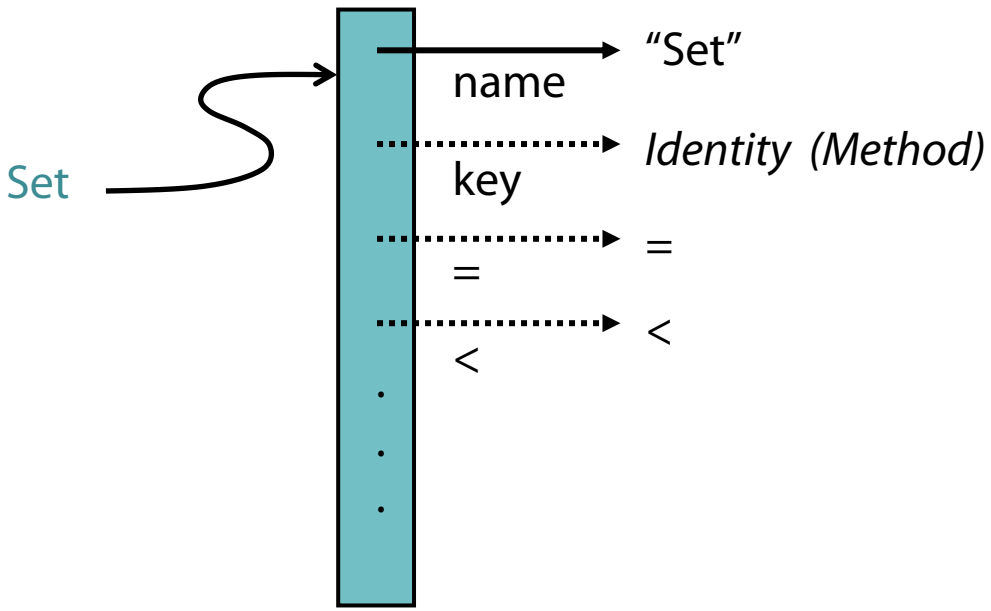
method key(S:Set)(x)

    return x

method =(S:Set)(x, y)

    return x = y

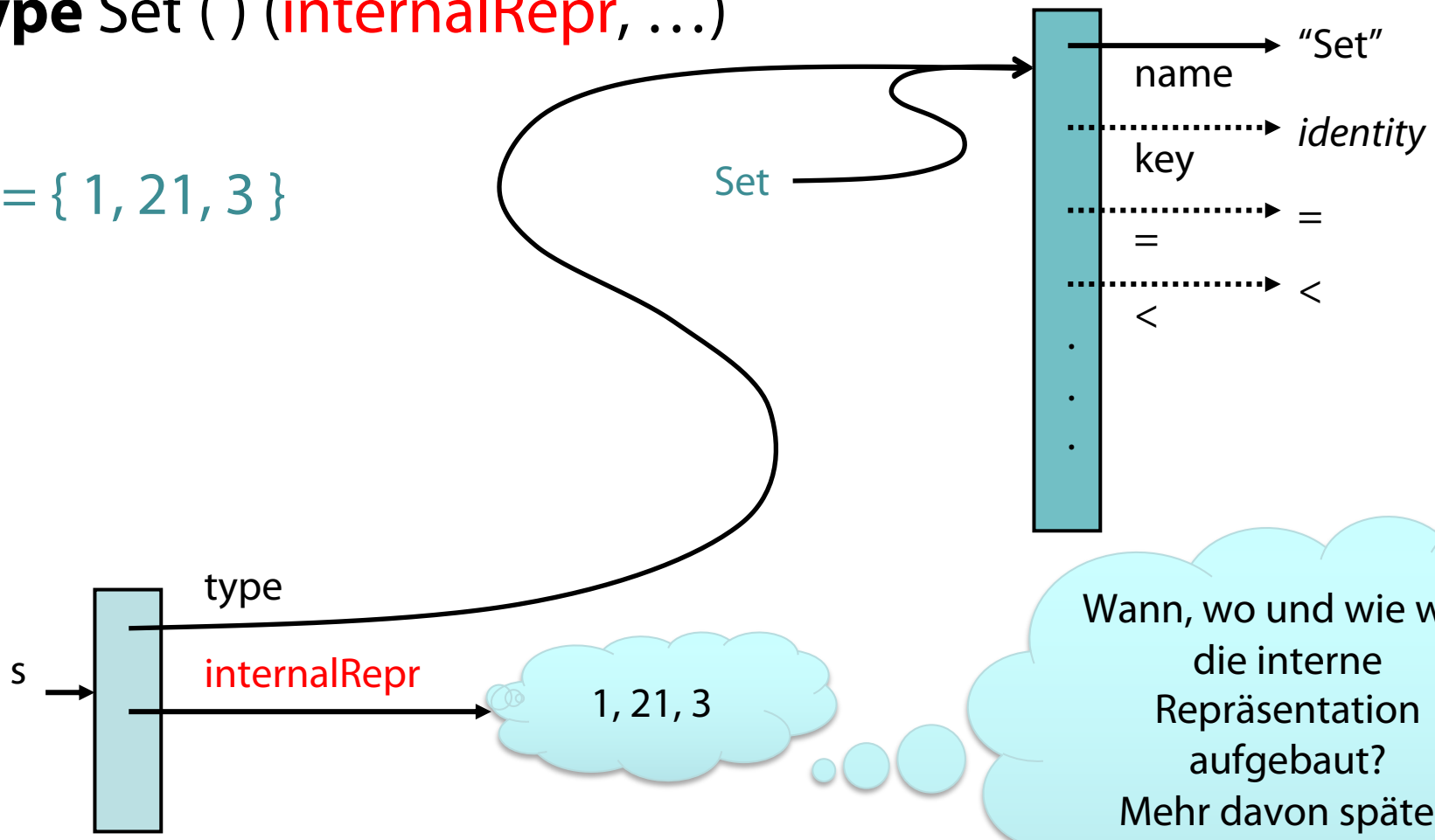
...



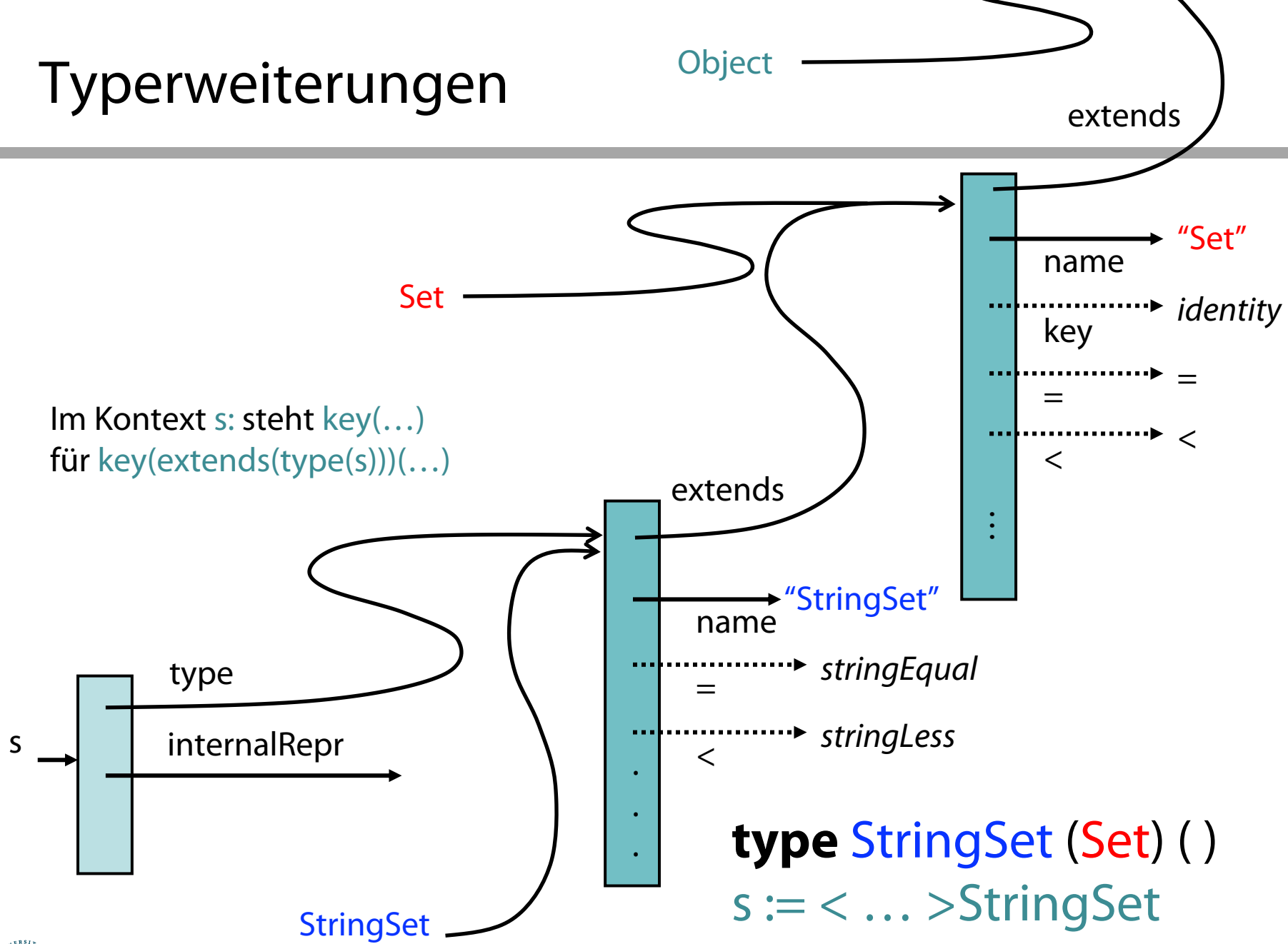
Typen und instanzspezifische Daten

type Set ( ) (internalRepr, …)

s := { 1, 21, 3 }



Typerweiterungen

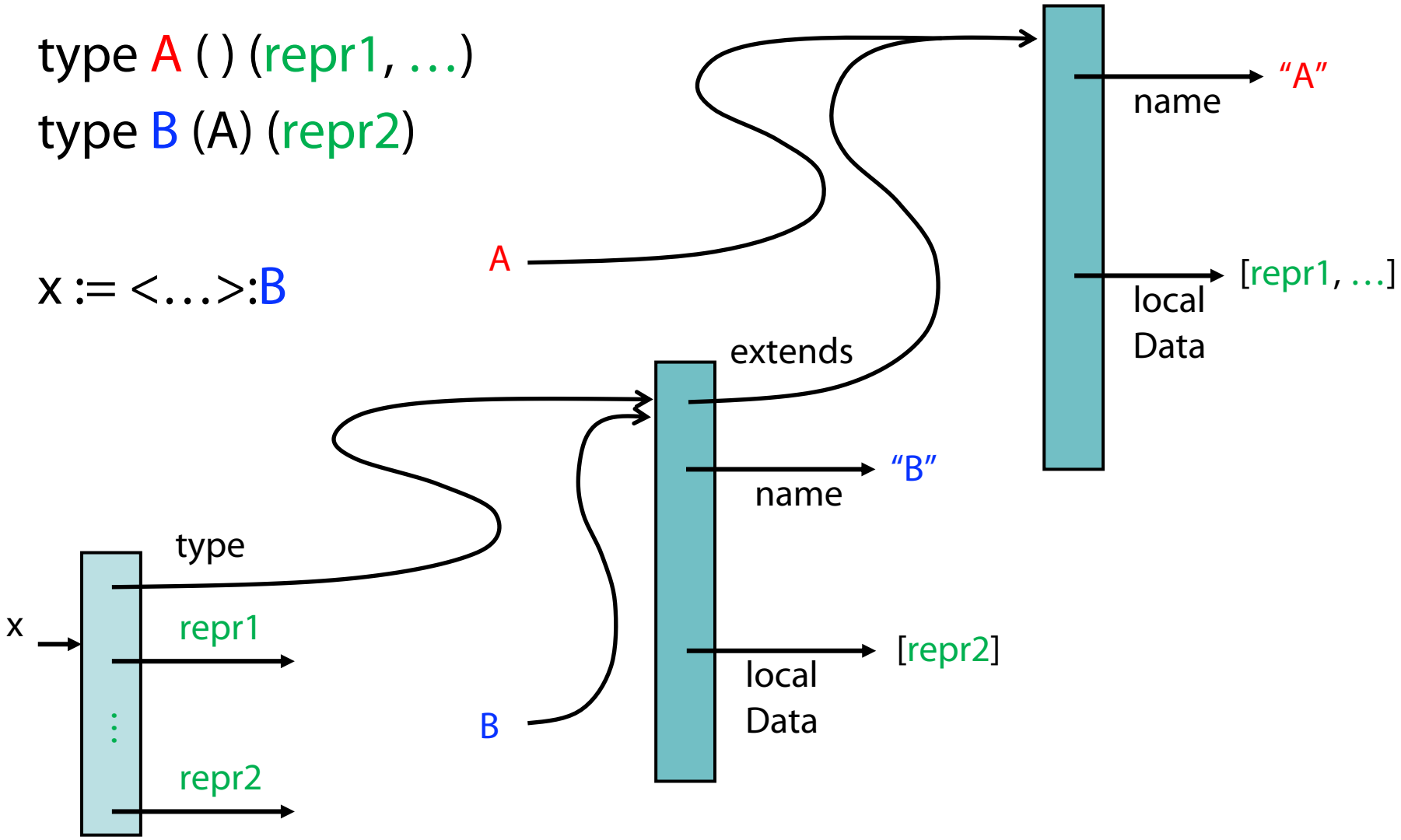


Typerweiterungen und Instanzen

type A ( ) (repr1, …)

type B (A) (repr2)

x := <…>:B



Mengen: API (1)

- Obligatorische Operationen
  - test(k, s:) testet, ob ein Element mit Schlüssel k in s enthalten ist liefert true oder false
  - search(k, s:)

    (1a) liefert das Element aus s, dessen Schlüssel k ist, oder nil

    (1b) liefert das Element aus s, dessen Schlüssel key minimal in s ist und für den key ≥ k gilt

    (2) liefert eine Menge von Elementen aus s, deren Schlüssel k ist
  - insert(x, s:) fügt das Element x in s ein
    (s wird ggf. modifiziert, wenn Element mit key(x) vorher in s enthalten, wird es aus s entfernt)
  - delete(k, s:) löscht Element x mit key(x) = k aus s
    (s wird modifiziert, wenn x enthalten war)

Mengen: API (2)

- Iterationsoperatoren
  - map(f, s:)
    - Wendet f auf jedes Element aus s an
      und gibt Ergebnisse als neue Menge zurück
    - Beispiel
      - **function** plus1(x) **return** x + 1
      - map(plus1, {42, 23, 17}) liefert {43, 24, 18}
  - fold(f, s:, init)
    - Wendet die Funktion f kaskadierend
      auf dem jeweils vorigen Wert, beginnend mit init, und jeweils alle Elemente in s an und liefert letzten Wert (bzw. init, wenn s leer)
    - Beispiel
      - **function** max(a, b) **if** a>b **then return** a **else return** b
      - fold(max, <4, 12, 24, 2>:Set, 0) liefert 24

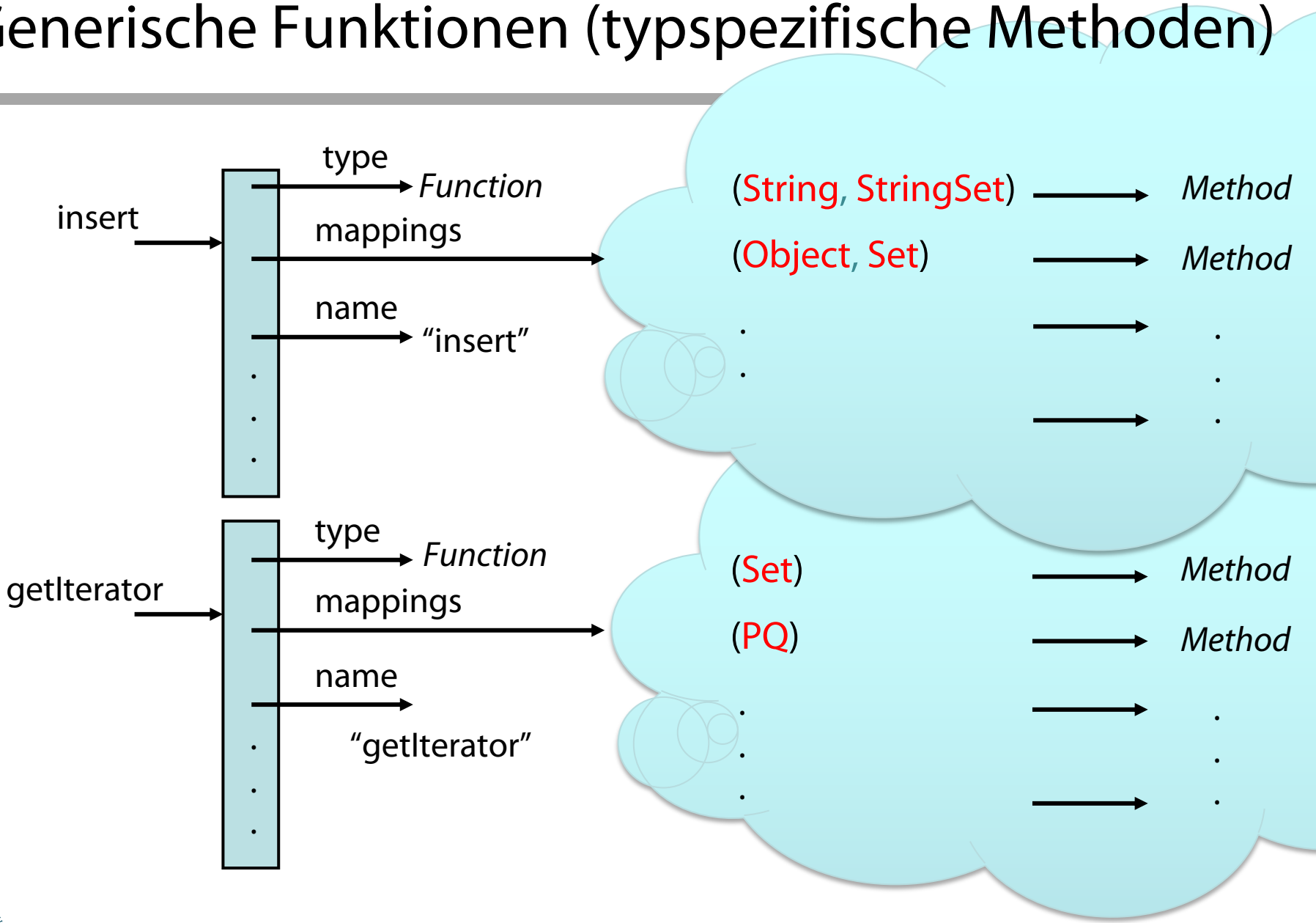
Mengen: API (3)

- Zusätzliche Operationen
  - setEqual(s1, s2)
  - union(s1, s2) (siehe auch merge bei PQs)
  - intersect(s1, s2)

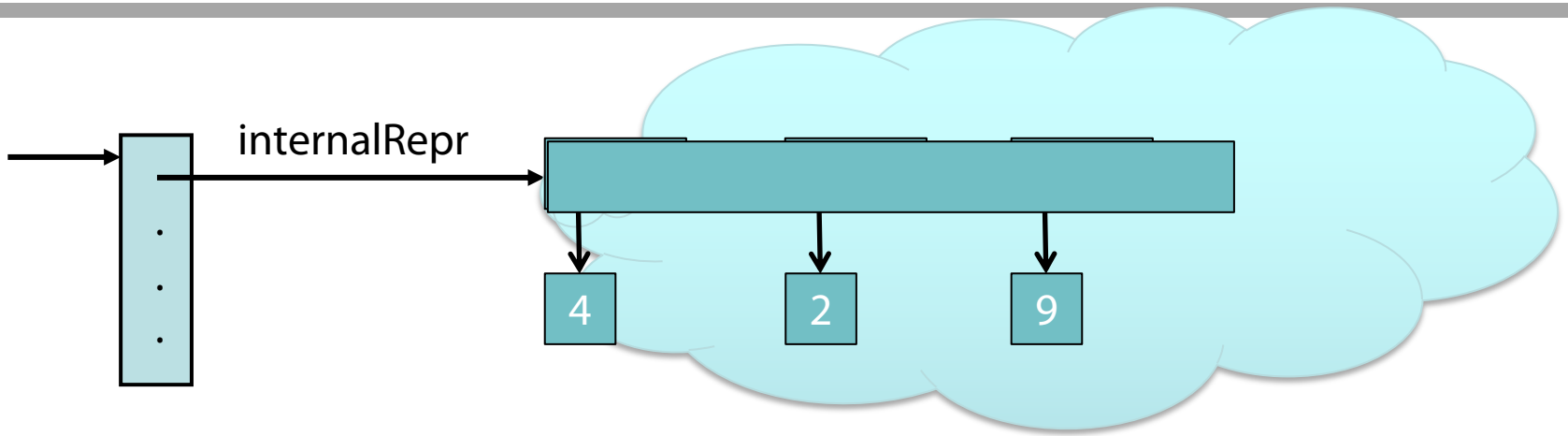
- Annahme
  - x := {“Rolly”, “Penny”}:StringSet
- Typspezifische Funktionen und Prozeduren?
  - **procedure** insert(str:String, s:StringSet) …
  - Aufruf mit insert(“Lucky”, x)
- Dann sollte auch möglich sein:
  - **function** getIterator(s:Set) …
  - Aufruf mit getIterator(x)

Sollte auch für StringSet-Instanz anwendbar sein. Mehr davon später.

Generische Funktionen (typspezifische Methoden)



Menge als ADT



1. Repräsentation als verkettete Liste?

2. Repräsentation als Array?

Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus:

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Suchstrukturen) gehalten von Christian Scheideler an der TUM
  http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

- „Algorithmen und Datenstrukturen“
  gehalten von Sven Groppe an der UzL

Repräsentation als verkettete Liste?

- **Günstigster Fall:** Element wird an 1. Stelle gefunden: $T_{min}(n) \in \Theta(1)$
- **Ungünstigster Fall:** Element wird an letzter Stelle gefunden (komplette Folge wurde durchlaufen): $T_{max}(n) \in \Theta(n)$
- **Durchschnittlicher Fall (Element ist vorhanden):**
  Annahme: kein Element wird bevorzugt gesucht:

$$T_{avg}(n) = \frac{1}{n} \times \sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n \times (n+1)}{2} = \frac{n+1}{2} \in \Theta(n)$$

- **Falls Misserfolg bei der Suche (Element nicht gefunden):**
  Es muss die gesamte Folge durchlaufen werden: $T_{fail}(n) \in \Theta(n)$

Selbstanordnende Listen

- **Idee:**
  - Ordne die Elemente bei der sequentiellen Suche so an, dass die Elemente, die am häufigsten gesucht werden, möglichst weit vorne stehen
    - Meistens ist die Häufigkeit nicht bekannt, man kann aber versuchen, *aus der Vergangenheit auf die Zukunft zu schließen*
- **Vorgehensweise:**
  - Immer wenn nach einem Element gesucht wurde, wird dieses Element weiter vorne in der Liste platziert

Strategien von selbstanordnenden Listen

- ***MF - Regel*, Move-to-front:**
  Mache ein Element zum ersten Element der Liste, wenn nach diesem Element erfolgreich gesucht wurde. Alle anderen Elemente bleiben unverändert.
- ***T - Regel*, Transpose:**
  Vertausche ein Element mit dem unmittelbar vorangehenden nachdem auf das Element zugegriffen wurde
- ***FC - Regel*, Frequency Count:**
  Ordne jedem Element einen Häufigkeitszähler zu, der zu Beginn mit 0 initialisiert wird und der bei jedem Zugriff auf das Element um 1 erhöht wird. Nach jedem Zugriff wird die Liste neu angeordnet, so dass die Häufigkeitszähler in absteigender Reihenfolge sind.

Beispiel selbstanordnende Listen, MF-Regel

- Beispiel (für Worst Case)

Zugriff	(resultierende) Liste	Aufwand in zugegriffenen Elementen
	7-6-5-4-3-2-1	
1	1-7-6-5-4-3-2	7
2	2-1-7-6-5-4-3	7
3	3-2-1-7-6-5-4	7
4	4-3-2-1-7-6-5	7
5	5-4-3-2-1-7-6	7
6	6-5-4-3-2-1-7	7
7	7-6-5-4-3-2-1	7

- Durchschnittliche Kosten: 7x7/7

Beispiel selbstanordnende Listen, MF-Regel

- Beispiel (für „beinahe“ Best Case)

Zugriff	(resultierende) Liste	Aufwand in zugegriffene Elemente
	7-6-5-4-3-2-1	
1	1-7-6-5-4-3-2	7
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1
1	1-7-6-5-4-3-2	1

- Durchschnittliche Kosten: $7 + 6 \times 1/7 \approx 1.86$

Beispiel selbstanordnende Listen, MF-Regel

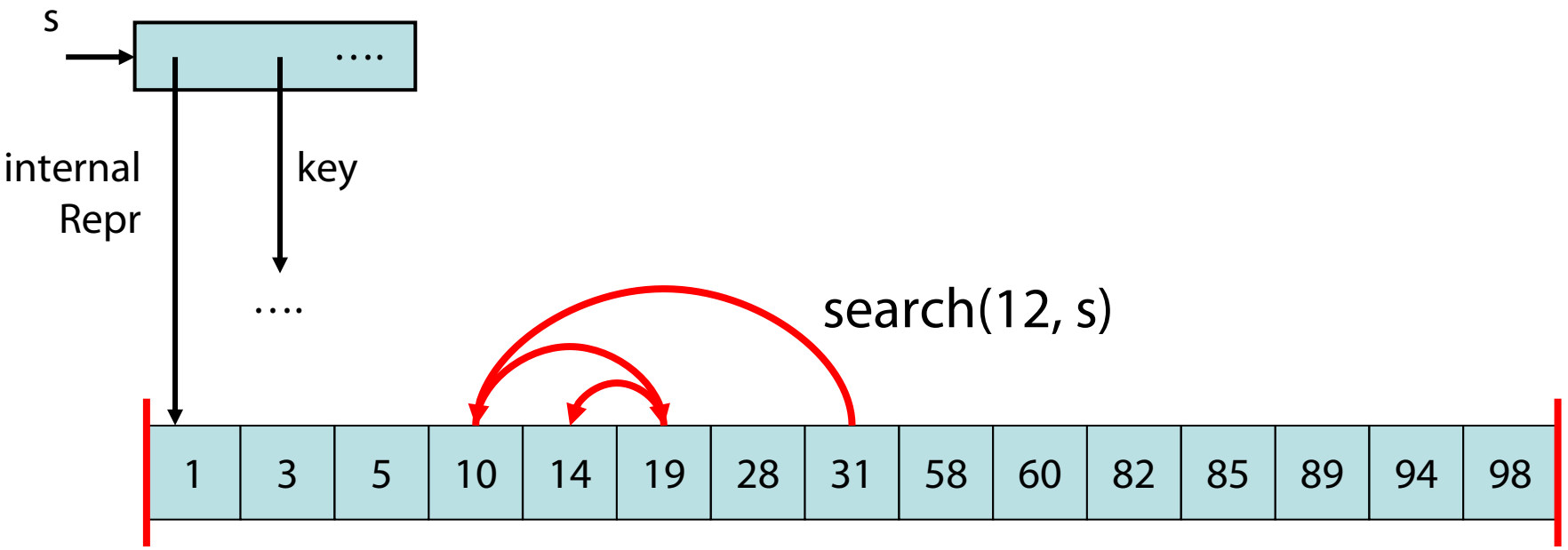
- **Feste Anordnung und naives Suchen** hat bei einer 7-elementigen Liste durchschnittlich den Aufwand:

$$\frac{1}{7} \times \sum_{i=1}^{7} i = \frac{1}{7} \times \frac{7 \times 8}{2} = 4$$

- Die *MF -Regel* kann also Vorteile haben gegenüber einer festen Anordnung
  - Dies ist insbesondere der Fall, wenn die Suchschlüssel stark gebündelt auftreten
  - Näheres zu selbstanordnenden Listen findet man im Buch von *Ottmann und Widmayer*

Repräsentation als Array?

Speichere Elemente in sortiertem Feld

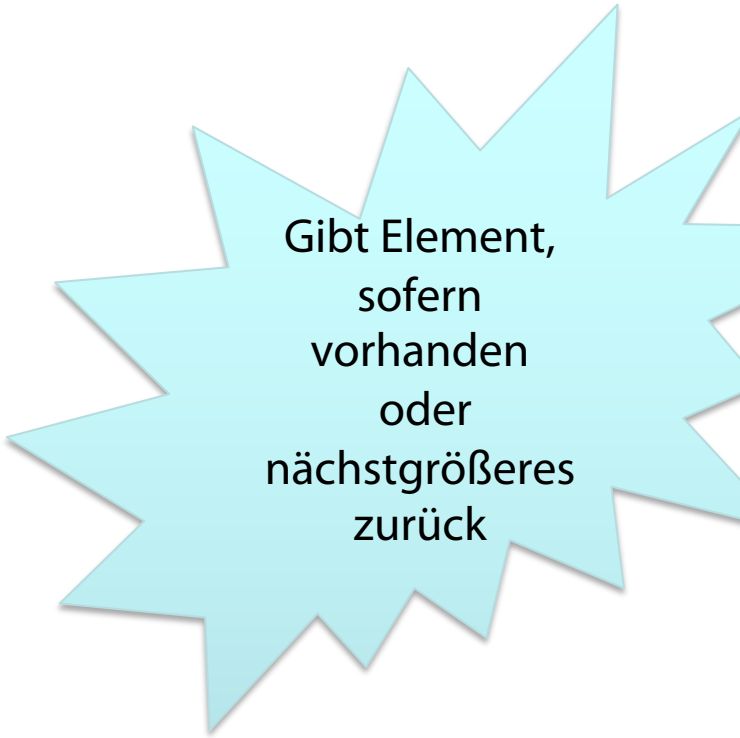


search: über binäre Suche (O(log n) Zeit)

Binäre Suche

Eingabe: Zahl x und eine Menge repräsentiert als sortiertes Feld

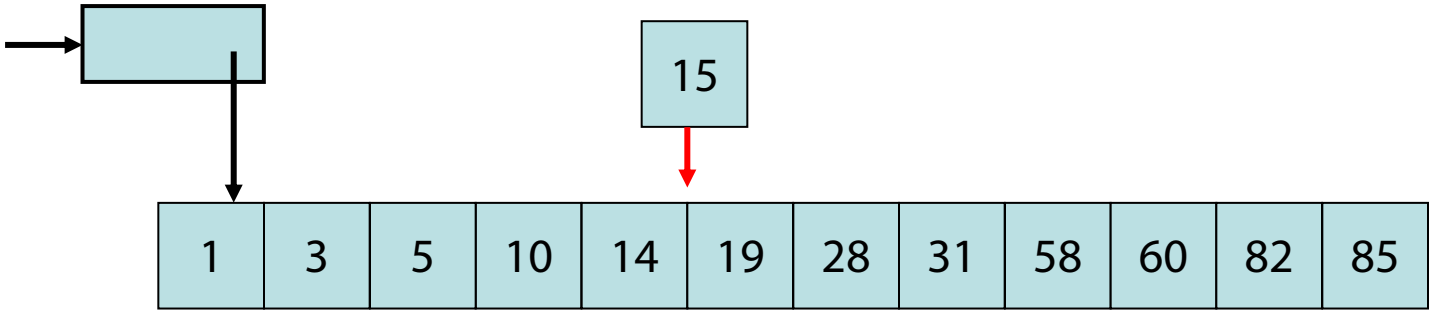
```
function search(k, s:)
  A := internalRepr(s)
  l := 1; r := length(A)
  while l < r do
    m:=(r+l) div 2
    if key(A[m]) = k then return A[m]
    if key (A[m]) < k then l:=m+1
    else r:=m
  return A[l]
```



Einfügen und Löschen bei Arrays

insert und delete Operationen:

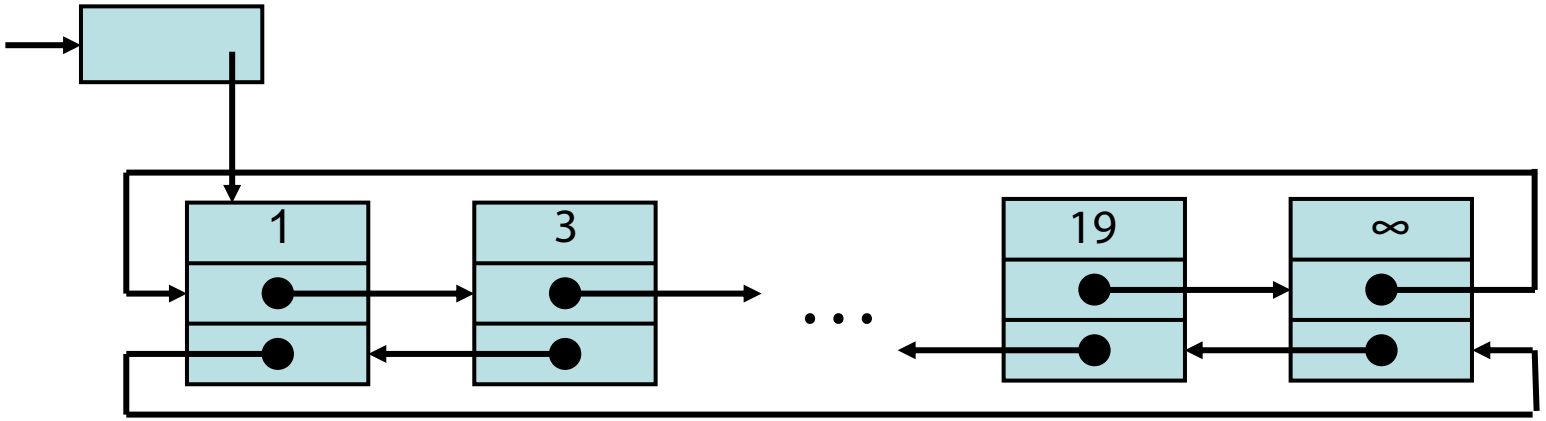
Sortiertes Feld schwierig zu aktualisieren!



Schlimmster Fall: $\theta(n)$ Zeit

Repräsentation als sortierte Liste?

Sortierte Liste (hier zyklisch und doppelt verkettet)



Problem: insert, delete und search kosten im schlimmsten Fall $\theta(n)$ Zeit

Einsicht: Wenn search effizient zu implementiert, dann auch alle anderen Operationen

Search gibt Element oder nächstgrößeres zurück

Wahl der Repräsentationsstruktur

- Betrachtete Alternativen
  - Einfach verkettete Liste
  - Sortiertes Array
  - (Zyklische) doppelt verkettete sortierte Liste

Welche Möglichkeiten ergäben sich durch eine zyklische Liste?

Was wäre die Folge bei einer einfach verketteten Liste bzgl. der Iterationsmöglichkeiten?

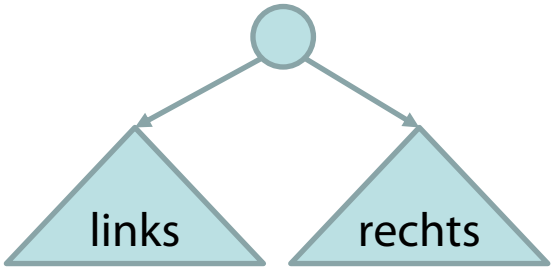
Motivation für Suche nach neuer Trägerstruktur

- Binäre Suche durchsucht Felder in O(log(n))
  - Einfügen neuer Elemente erfordert Verschiebung und ggf. ein größeres Feld und Umkopieren der Elemente
- Einfügen in Listen in konstanter Zeit
  - Zugriffe auf Elemente jedoch sequentiell
- (Verzeigerter) Baum
  - Zum Suchen verwendbar
  - Einfügen: erst Suche, dann an richtiger Position einfügen
  - Vorwärts- und Rückwärtsiteration?

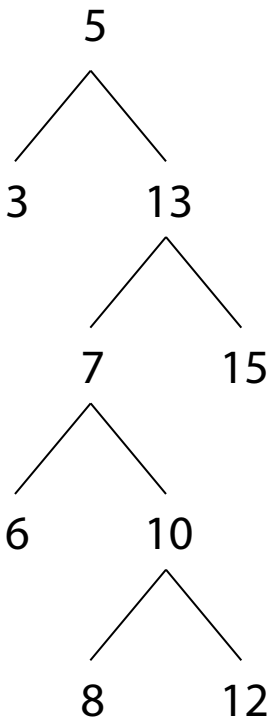
Binäre Suchbäume - Definition

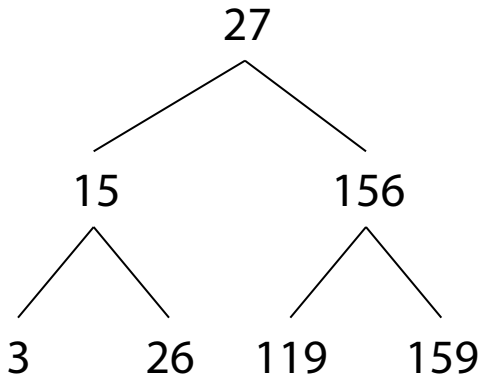
- Ein binärer Suchbaum
  1. ist ein Binärbaum, und
  2. zusätzlich muss für jeden seiner Knoten gelten, dass das im Knoten *gespeicherte Element*
     a) **≥** ist als alle *Elemente* im *linken Unterbaum*
     b) **<** ist als alle *Elemente* im *rechten Unterbaum*

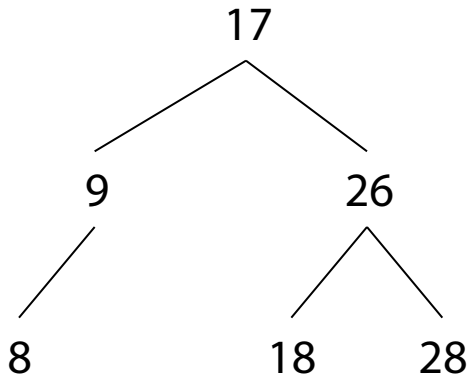
Unterschied zum Heap?



Beispiele für binäre Suchbäume



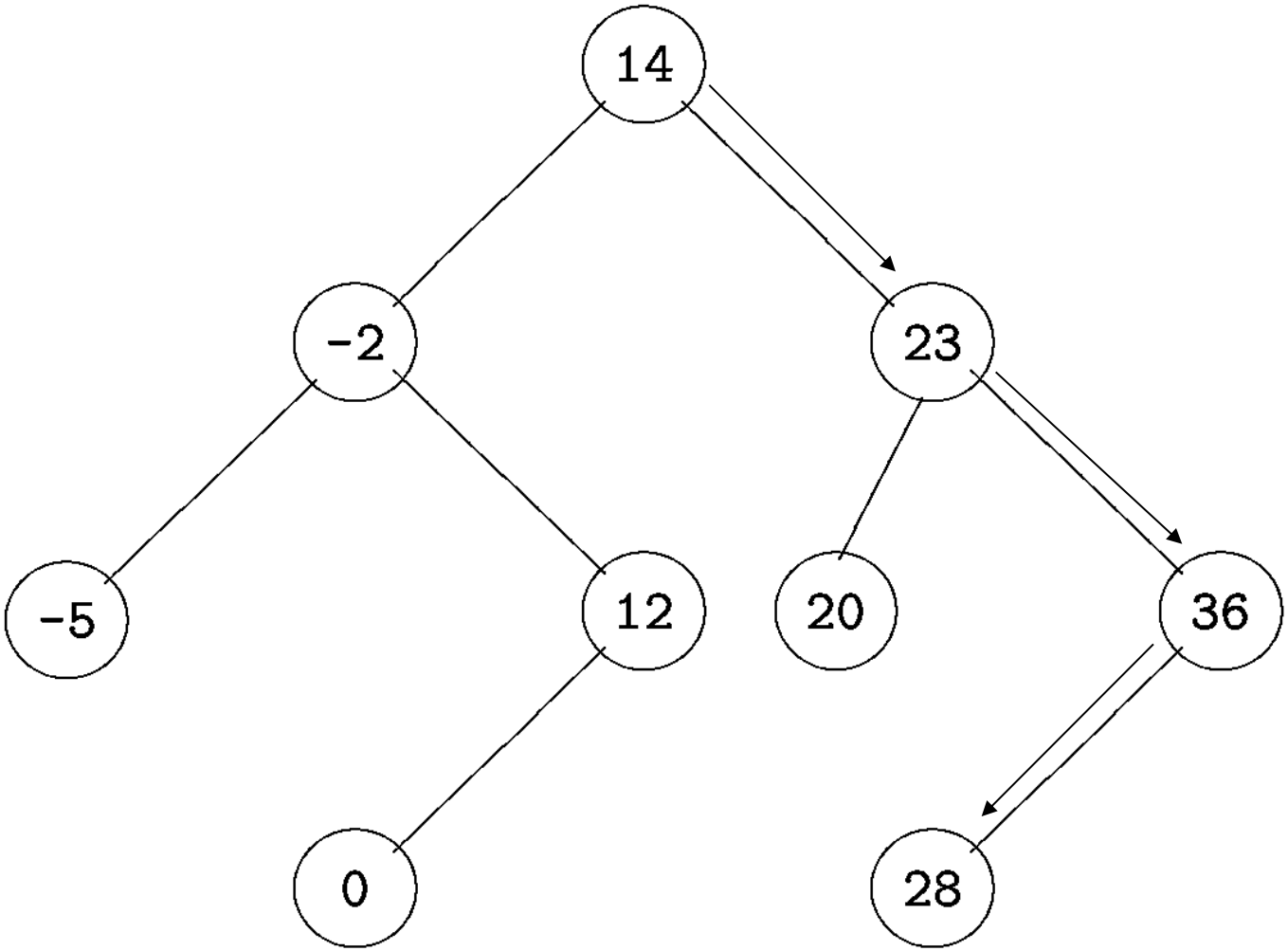




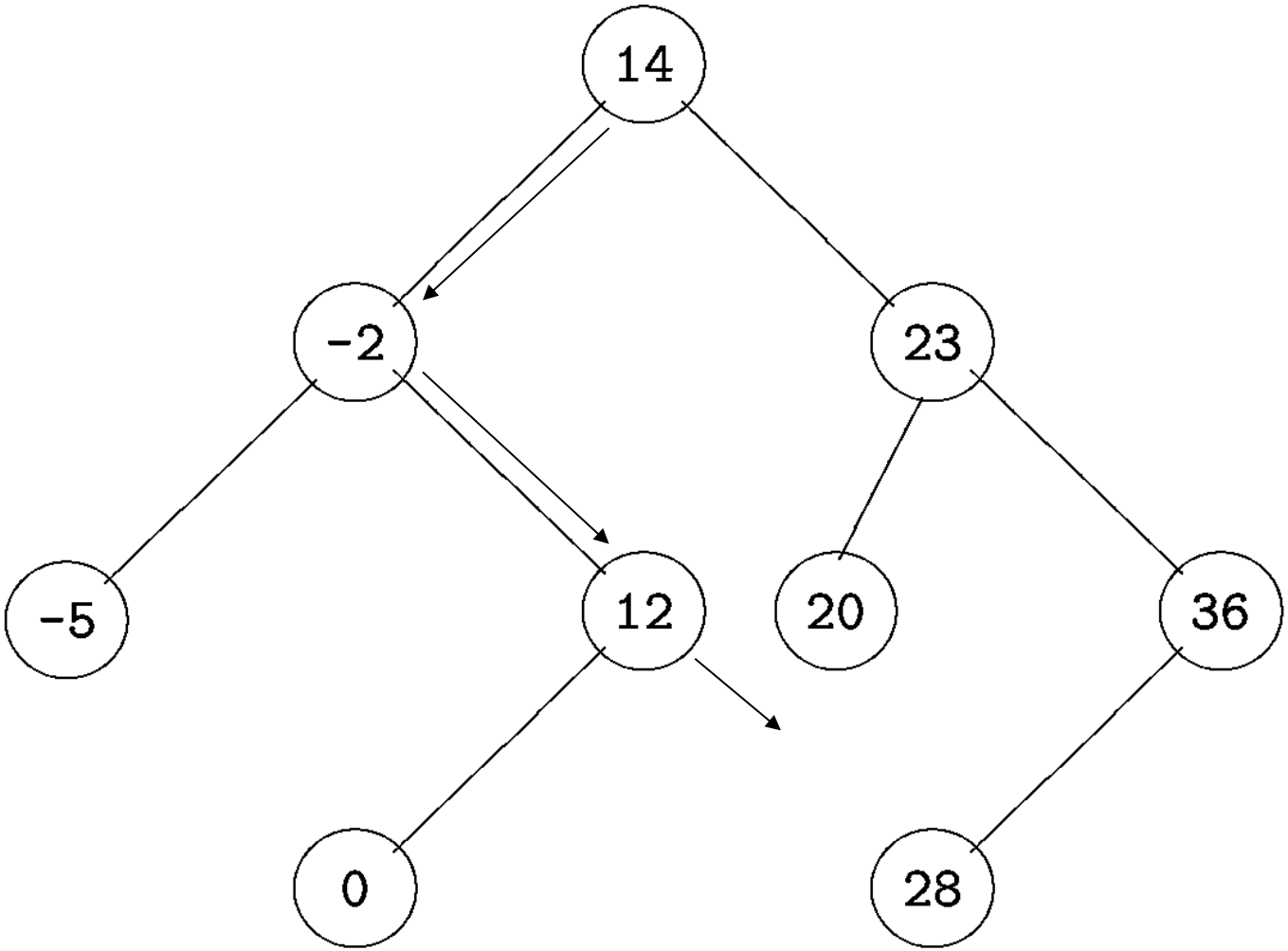
Suche im binären Suchbaum

- Suche nach einem Element im binären Suchbaum:
  - Baum ist leer:
    - Element nicht gefunden
  - Baum ist nicht leer:
    - Wurzelelement ist gleich dem gesuchten Element:
      - Element gefunden
    - Gesuchtes Element ist kleiner als das Wurzelelement:
      - Suche im linken Unterbaum rekursiv
    - Gesuchtes Element ist größer als das Wurzelelement:
      - Suche im rechten Unterbaum rekursiv

Suche nach 28



Suche nach 13

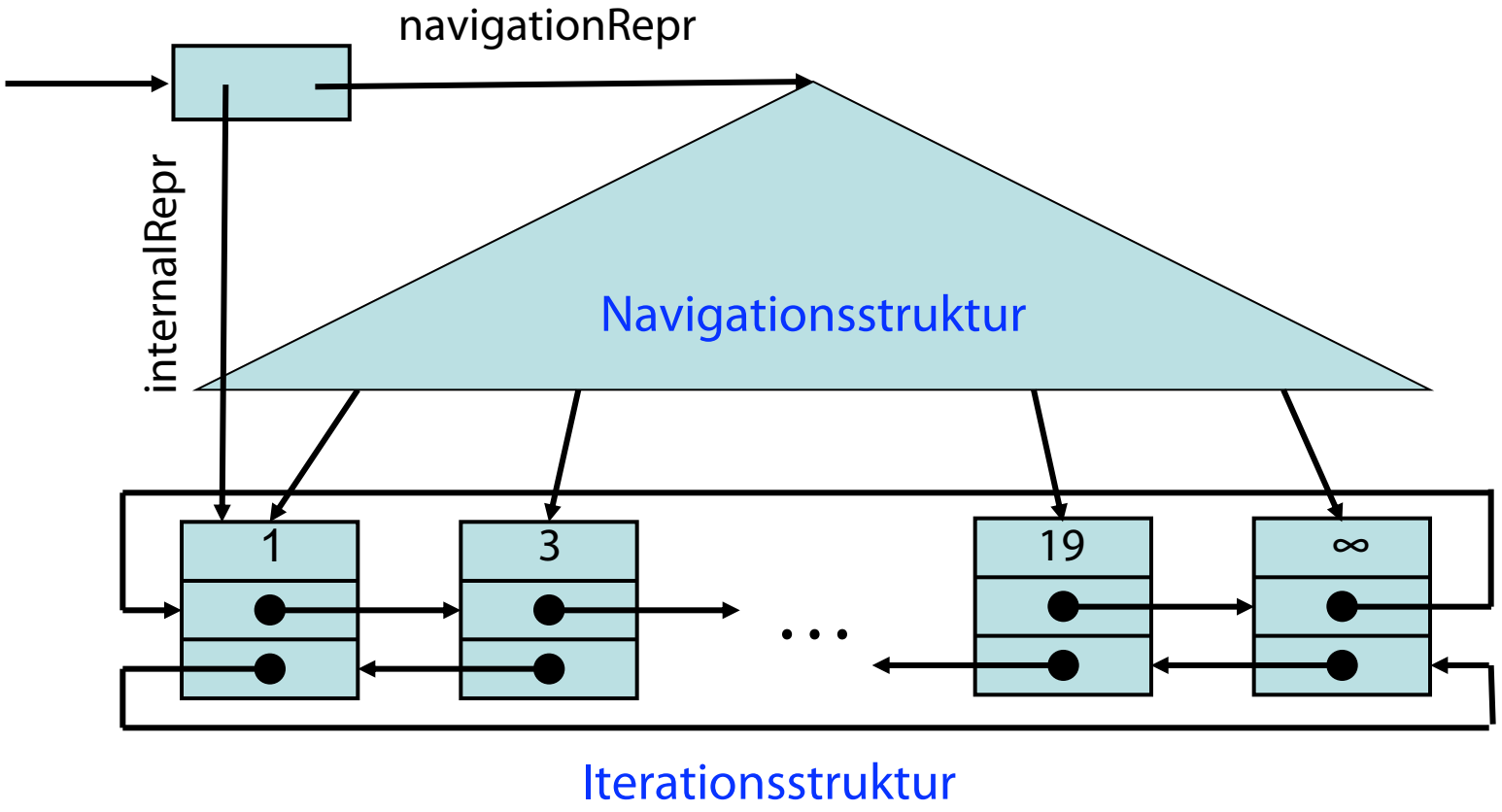


Binärer Suchbaum

- Wie kann man Iteratoren realisieren?
- getIterator(s) bzw. getIterator(s, fromKey)

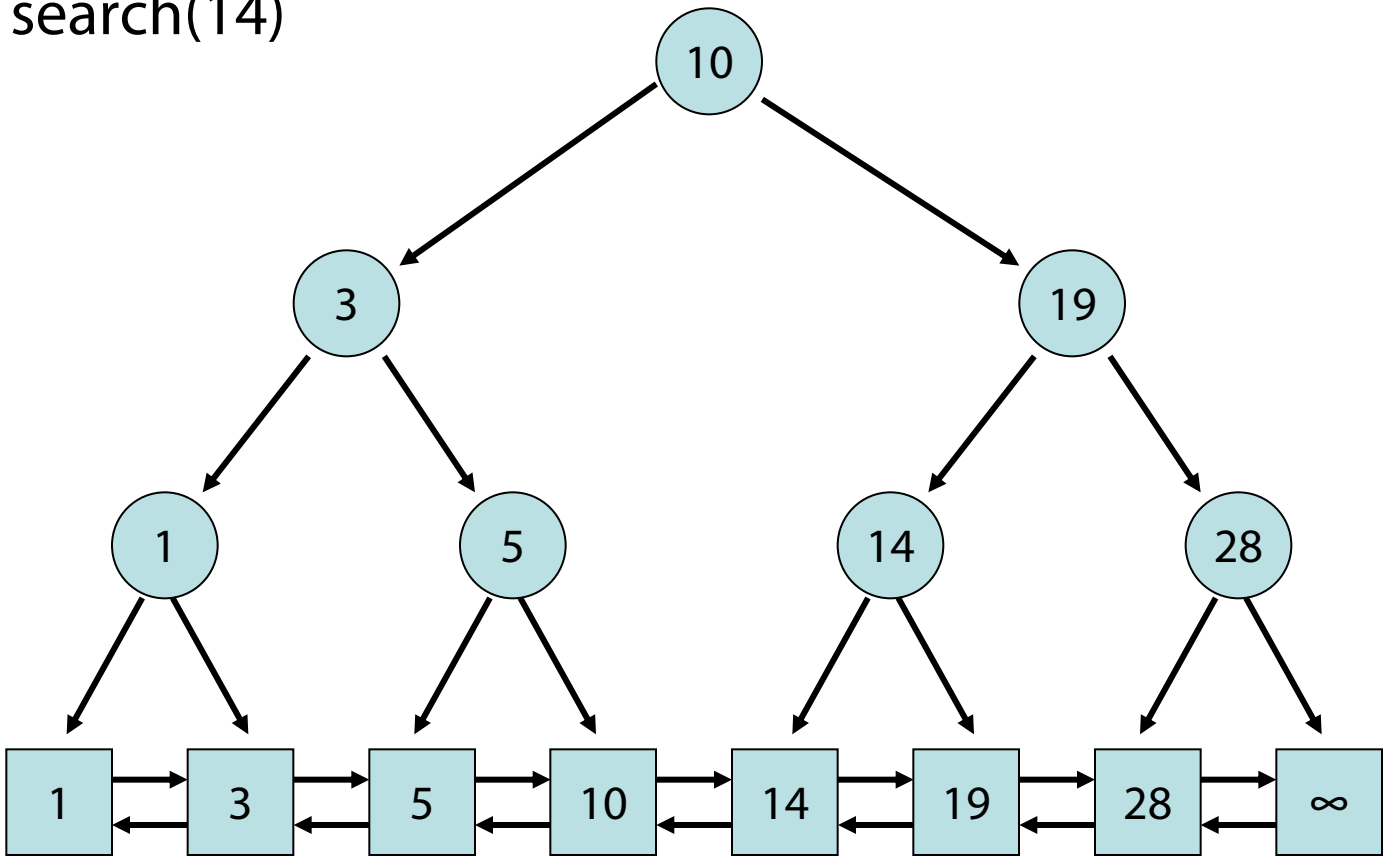
Suchstruktur

Idee: Baum als Navigationsstruktur (nur Schlüsselwerte), die search effizient macht, plus doppelt verkettete zyklische Liste als Iterationsstruktur und zum einfachen Einfügen



Binärer Suchbaum (ideal)

search(14)

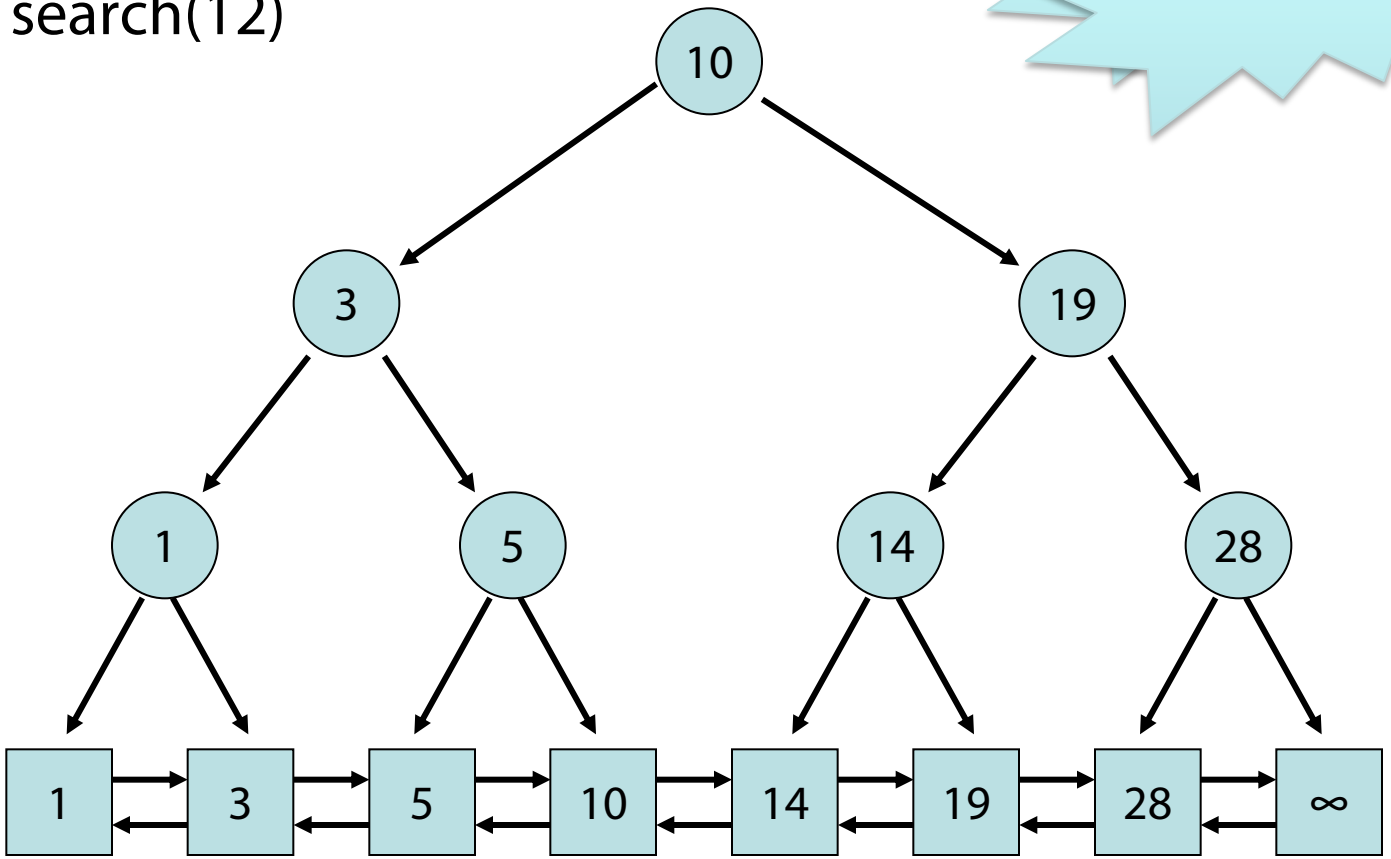


Zyklus nicht gezeigt

Binärer Suchbaum (ideal)

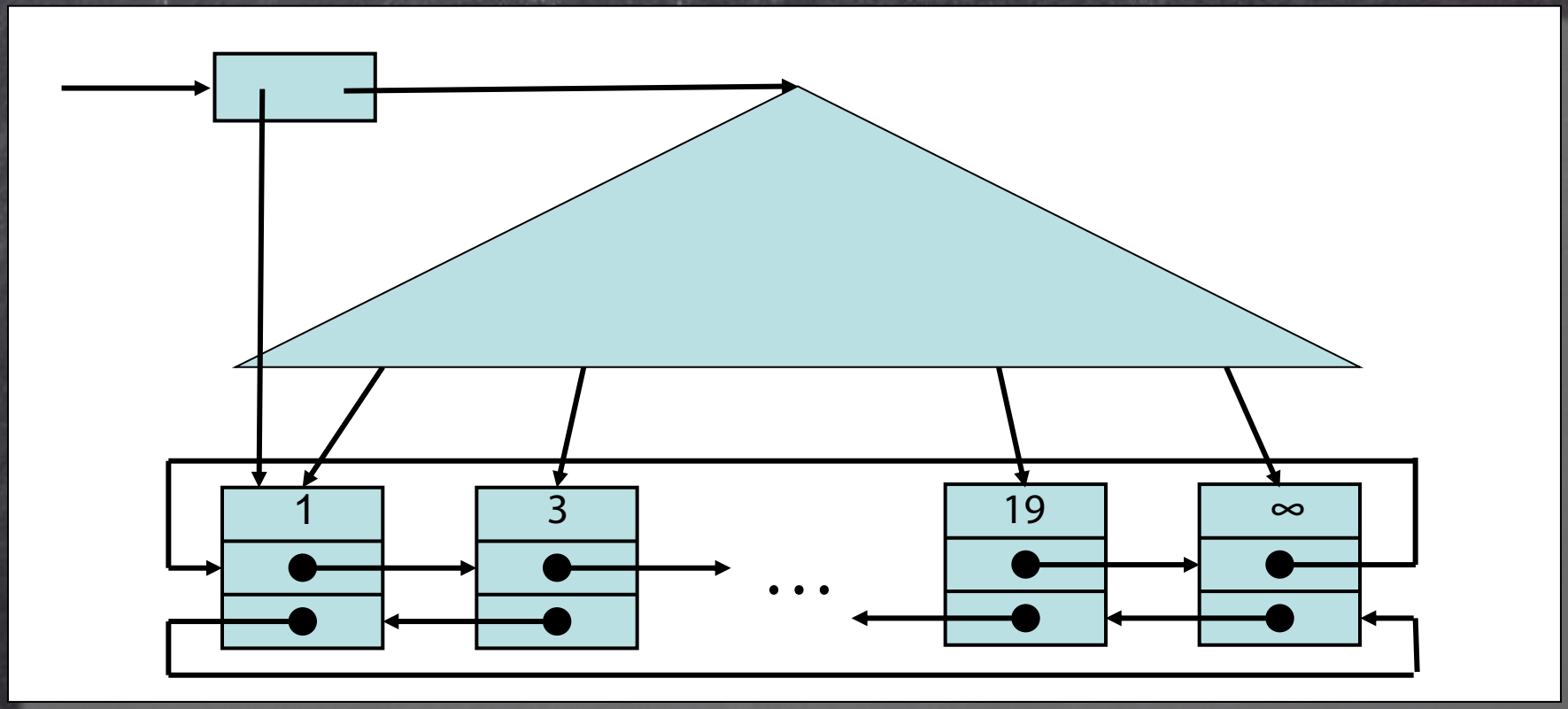
Geordnete Menge?

search(12)



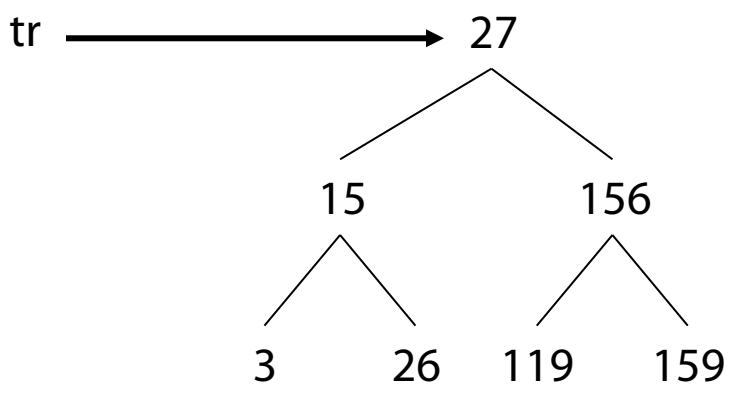
Aufgabe: Iteration über alle Elemente?

- Wie Elemente addieren?
- Wie Maximum/Minimum bestimmen?
- K-Kleinstes Element?



Aufgabe: Iteration auch über Baumknoten?

- Wie Elemente addieren?
- Wie Maximum/Minimum bestimmen?



fold(+, tr, 0)
→ 505

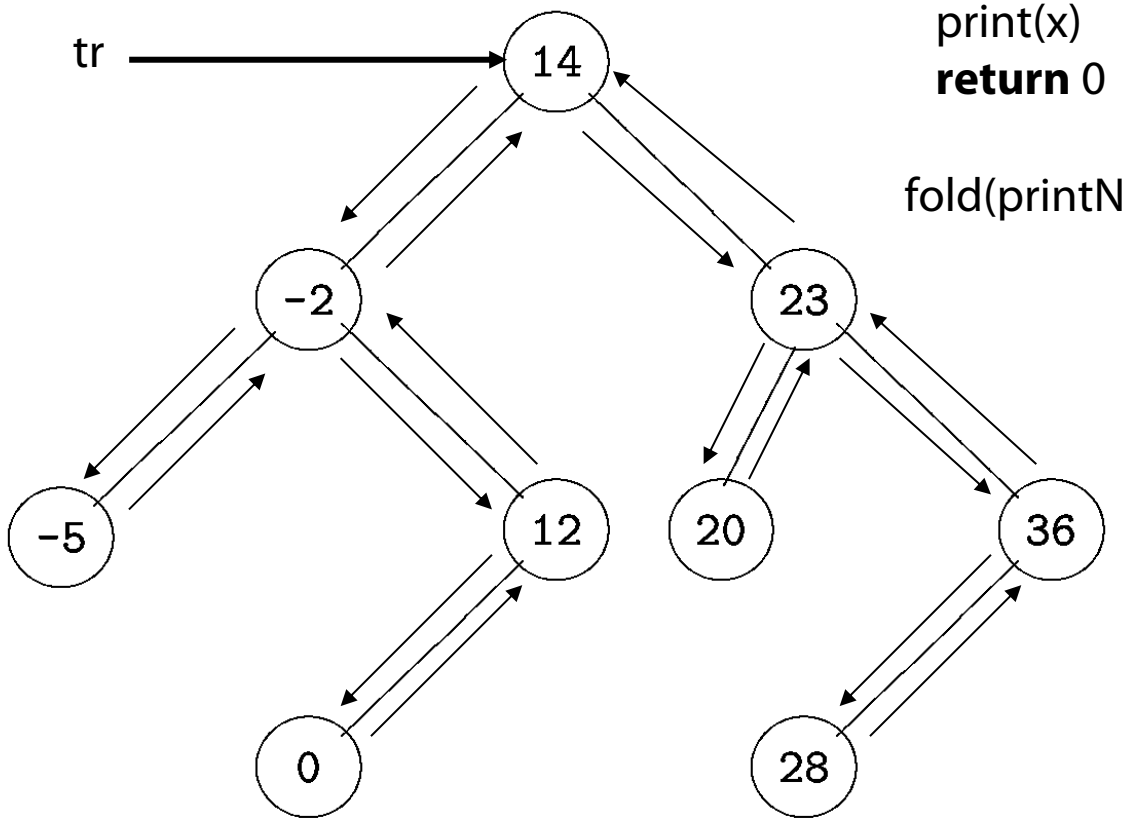
```
function max (x, y)
  if  x > y then
    return x
  else return y
```

fold(max, tr, 0)
→ 159

```
function min (x, y)
  if  x > y then
    return y
  else return x
```

fold(min, tr, ∞)
→ 3

Inorder-Ausgabe ergibt Sortierreihenfolge




```
function printNode(ignore, x)
  print(x)
  return 0

fold(printNode, tr, 0)
```

Können Sie fold für Bäume so realisieren, dass das klappt?

-5 -2 0 12 14 20 23 28 36

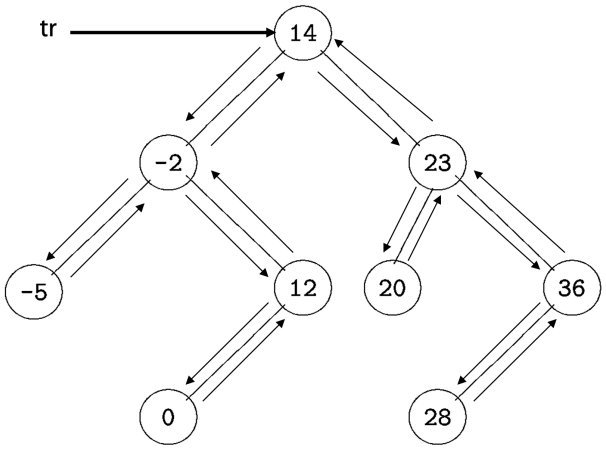
Aufgabe: Fold für Bäume

```
function fold(f, tr:TreeNode, init)
  if mtTree?(tr) then
    return init
  if leaf?(tr) then
    return f(init, key(tr))
  if leftExists?(tr) then
    x := f(  fold(f, left(tr), init),  key(tr) )
    if rightExists?(tr) then
      return fold(f, right(tr), x)
    else return x
  else // linker Nachfolger existiert nicht, rechter ja, sonst leaf
    return fold( f, right(tr), f(init, key(tr)) )
```

key(tr) liefert Knotenbeschriftung als Komponente von einem TreeNode

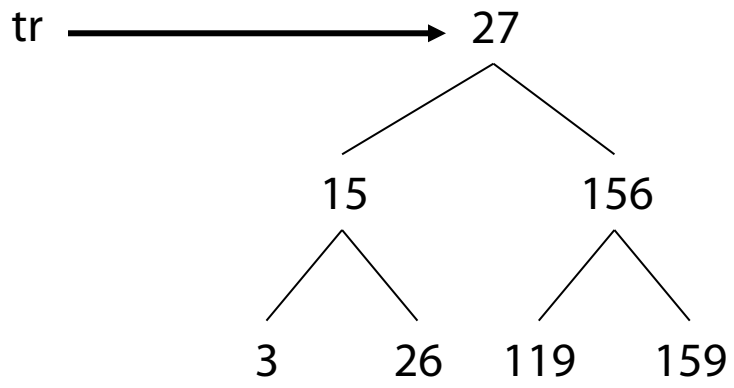
```
function printNode(ignore, x)
  print(x)
  return 0

fold(printNode, tr, 0)
```



Aufgabe: Iteration auch über Baumknoten?

- K-Kleinstes Element bestimmen?
- Mit fold?



Zusammenfassung

- Abstrakte Datentypen
  - Typen
  - Instanzen
  - Generische Funktionen
  - Methoden (in Instanzen, in Typen, in generischen Funktionen)
- Repräsentation von Mengen
  - Liste
    - Ggf. selbstadaptierend
  - Array
  - Baum
  - Kombiniert
    - Baum zur Navigation
    - Doppelt verkettete zyklische Liste zur Iteration in beide Richtungen