
Suchen abstrahieren und konkretisieren

Schlagwörter	Suchproblem, Suchraum, lineare Suche, binäre Suche, Programmierstil, Programmumformung, Programmverifikation
Motivation	Suchen von Daten in Datenmengen ist eine ständig wiederkehrende Programmieraufgabe, die jeder Informatiker beherrschen muss. Wer „Ich kann <i>Java!</i> “ sagt, muss wenigstens einen linearen strukturierten Suchalgorithmus schreiben können.
Widmung	Dieser Text ist Tobias Ableitner gewidmet. Tobias hält Implementationssprachen für präziser als Pseudosprachen und bevorzugt deshalb erstere. Dagegen nutzt dieser Text Pseudosprachen, um Problemlösungen auf hohem Abstraktionsniveau zu formalisieren, präzisieren, transformieren, verifizieren, ohne den Ballast unwesentlicher konkreter Details von Implementationssprachen mitzuschleppen.
Lernziele	Der Leser lernt Suchprobleme und ihre Lösung in verschiedenen Abstraktionsebenen, Formen, Programmierstilen und Notationen kennen, sodass er nach dem Studium des Texts zusichern kann: „Ich beherrsche das Programmieren von Suchalgorithmen!“ Zudem lernt er Lektionen zum Finden von Abstraktionen, vertraglichen Spezifizieren von Schnittstellen, Umformen von Programmen und Beweisen ihrer Korrektheit.

Inhalt

1	Mathematisch	1
2	Funktional	3
3	Imperativ rekursiv	4
4	Imperativ iterativ	6
5	Objektorientiert	9
6	Vertraglich spezifiziert	12
7	Linear	13
8	Binär	17
9	Beschleunigt	24
10	Alternativ abstrakt und konkret	28
11	Gefunden, gelesen, geprüft	32
12	Implementiert	43
A	Literatur	45

1

Mathematisch

Aufgabe	Das Problem, ein Element in einer Menge zu suchen, und seine Lösung soll in verschiedenen mathematischen und informatischen Abstraktionsebenen und Notationen dargestellt werden. Die allgemeine Vorgehensweise ist: Erst die Problemlösung auf hohem Abstraktionsniveau formalisieren, präzisieren und spezifizieren, dann möglichst spät konkretisieren, detaillieren und implementieren. Wir beginnen mit einer Darstellung in mengentheoretischen und prädikatenlogischen Begriffen.
Leitlinie	

Rekursive Definition

Ein **Suchraum** ist eine Menge S mit diesen Eigenschaften:

- (1) S ist entweder leer, $S = \emptyset$, oder
- (2) es gibt ein **aktuelles Element** $a_S \in S$ und
- (3) S ist auf einen kleineren Suchraum S' ohne a_S **reduzierbar**, $S' \subseteq S \setminus \{a_S\}$.

Gegeben sei ein Suchraum S und ein darauf definiertes Prädikat $p, p : S \rightarrow \{\text{falsch, wahr}\}$. Dann hat das **allgemeine Suchproblem** zwei Formen:

Existenzproblem

- (4) Gibt es ein Element $e \in S$, für das $p(e)$ gilt?

Allproblem

- (5) Gilt $p(e)$ für alle Elemente $e \in S$?

Im Folgenden beschränken wir uns auf die **Existenzform** (4) des allgemeinen Suchproblems. (Der Leser möge die folgenden Ausführungen selbst auf die **Allform** (5) übertragen.) Hier darf beim Reduzieren von S auf S' kein e mit $p(e)$ entfernt werden, d.h. S wird so in S' und $S \setminus S'$ zerlegt, dass $p(e)$ für kein $e \in S \setminus S'$ gilt. $S = (S \setminus S') \cup S'$ vorausgesetzt gilt formal:

$$\begin{aligned} \exists e \in S : p(e) &\equiv \exists e \in (S \setminus S') \cup S' : p(e) \\ &\equiv \exists e \in S \setminus S' : p(e) \vee \exists e \in S' : p(e) \\ &\equiv \neg \forall e \in S \setminus S' : \neg p(e) \vee \exists e \in S' : p(e) \\ &\equiv \forall e \in S \setminus S' : \neg p(e) \Rightarrow \exists e \in S' : p(e). \end{aligned}$$

Verbal: Wenn aus dem Suchraum nur nicht gesuchte Elemente entfernt werden, dann kommen gesuchte Elemente höchstens im reduzierten Suchraum vor. Damit lässt sich die Lösung des Existenzproblems als Rekursionsformel in zwei Formen darstellen. Die erste Form benötigt die kurz auswertende Konjunktion „und dann“, die zweite die kurz auswertende Disjunktion „oder sonst“, d.h. wir ergänzen die Logik um diese programmiersprachlichen nichtkommutativen logischen Operationen.

Rekursionsformel 1

Existiert Element mit Eigenschaft?

$$\exists e \in S : p(e) \Leftrightarrow \begin{cases} \text{falsch} & \text{falls } S = \emptyset & \text{-- nicht gefunden} \\ \text{wahr} & \text{falls } S \neq \emptyset \text{ und dann } p(a_S) & \text{-- gefunden} \\ \forall e \in S \setminus S' : \neg p(e) \Rightarrow & \text{sonst} & \text{-- reduzieren und} \\ \exists e \in S' : p(e) & & \text{-- weitersuchen!} \end{cases}$$

Rekursionsformel 2

Existiert Element mit Eigenschaft?

$$\exists e \in S : p(e) \Leftrightarrow \begin{cases} S \neq \emptyset & \text{falls } S = \emptyset \text{ oder sonst } p(a_S) & \text{-- nicht gefunden oder gefunden} \\ \forall e \in S \setminus S' : \neg p(e) \Rightarrow & \text{sonst} & \text{-- reduzieren und weitersuchen!} \\ \exists e \in S' : p(e) & & \end{cases}$$

Die Antwort auf die Frage (4) lautet also entweder „nein, alle $e \in S$ wurden geprüft und keines erfüllt $p(e)$ “, oder „ja, nämlich a_S erfüllt $p(a_S)$ “. Im Folgenden beschränken wir uns (abweichend von unserer Leitlinie früh konkretisierend) auf ein spezielles Prädikat. Gegeben sei ein Element $x \in G$ einer Grundmenge $G \supseteq S$, mit dem das Prädikat

Konkretes Prädikat

$$p_x(e) \quad :\Leftrightarrow \quad e = x$$

definiert sei. Damit lauten die zwei Formen dieses **speziellen Suchproblems**:

- (6) Ist $x \in S$?
- (7) Ist $S = \{x\}$?

Wir beschränken uns wieder auf die Existenzform (6). (Der Leser möge die folgenden Ausführungen zum speziellen Suchproblem selbst auf das allgemeine Suchproblem

verallgemeinern.) Hier darf beim Reduzieren von S auf S' nicht das gesuchte Element x entfernt werden. Wieder $S = (S \setminus S') \cup S'$ vorausgesetzt gilt formal:

$$x \in S \equiv x \in (S \setminus S') \cup S' \equiv x \in S \setminus S' \vee x \in S' \equiv x \notin S \setminus S' \Rightarrow x \in S'.$$

Damit lässt sich die Lösung des speziellen Suchproblems (6) als spezielle Rekursionsformel in zwei Formen darstellen.

Rekursionsformel 3

Enthält der Suchraum das gesuchte Element?

$$x \in S \Leftrightarrow \begin{cases} \text{falsch} & \text{falls } S = \emptyset & \text{-- nicht gefunden} \\ \text{wahr} & \text{falls } S \neq \emptyset \text{ und dann } a_S = x & \text{-- gefunden} \\ x \notin S \setminus S' \Rightarrow x \in S' & \text{sonst} & \text{-- weitersuchen!} \end{cases}$$

Rekursionsformel 4

Enthält der Suchraum das gesuchte Element?

$$x \in S \Leftrightarrow \begin{cases} S \neq \emptyset & \text{falls } S = \emptyset \text{ oder sonst } a_S = x & \text{-- nicht gefunden} \\ & & \text{-- oder gefunden} \\ x \notin S \setminus S' \Rightarrow x \in S' & \text{sonst} & \text{-- weitersuchen!} \end{cases}$$

2

Funktional

Wir nähern nun die Notation der Lösungen des speziellen Suchproblems schrittweise programmiersprachlichen Notationen an. Im ersten Schritt formalisieren wir die Fallunterscheidungen der Rekursionsformeln 3 und 4 zu bedingten Ausdrücken.

Rekursionsformel 5

Zweifach bedingter Ausdruck: Enthält der Suchraum das gesuchte Element?

```
x ∈ S ⇔ if S = ∅ then
    false
elseif aS = x then
    true
else
    x ∈ S'
end
```

Rekursionsformel 5 benötigt einen zweifach bedingten Ausdruck, kann aber auf die kurz auswertenden Operationen verzichten.

Rekursionsformel 6

Einfach bedingter Ausdruck: Enthält der Suchraum das gesuchte Element?

```
x ∈ S ⇔ if S = ∅ or else aS = x then
    S ≠ ∅
else
    x ∈ S'
end
```

Rekursionsformel 6 benötigt nur einen einfach bedingten Ausdruck, aber zusätzlich die kurz auswertende Disjunktion **or else**. Gegenüber Rekursionsformel 5 ist nachteilig, dass die Bedingung $S = \emptyset$ nochmal, und zwar negiert als $S \neq \emptyset$ vorkommt. Der Nachteil hat zwei Aspekte: Der Programmierer kann beim Duplizieren der Bedingung Fehler begehen, der Programmablauf braucht mehr Rechenzeit.

Ersetzen wir in Rekursionsformel 6 die Disjunktion gemäß

$$S = \emptyset \text{ or else } a_S = x \quad \equiv \quad \text{if } S = \emptyset \text{ then true else } a_S = x \text{ end}$$

durch den bedingten Ausdruck, so erhalten wir die nicht verständlichere Rekursionsformel 7, aus der sich durch „Ausrechnen“ wieder Rekursionsformel 5 ergibt.

Rekursionsformel 7
Umgeformte
Rekursionsformel 6

```

 $x \in S \Leftrightarrow$  if  $S = \emptyset$  then true else  $a_S = x$  end then
            $S \neq \emptyset$ 
           else
            $x \in S'$ 
           end

```

Im zweiten Schritt ordnet Tabelle 1 den mengentheoretischen Notationen Funktionen einer funktionalen Pseudosprache zu.

Tabelle 1
Mathematische und
informatische
Notationen

Mengentheoretisch	Programmiersprachlich funktional
$x \in S$	has (S, x)
$S = \emptyset$	is_empty (S)
a_S	item (S)
S'	reduced (S)

Die Ersetzung der Notationen in den Rekursionsformeln 5 und 6 gemäß Tabelle 1 liefert die Programme 1 und 2. Das Schlüsselwort **is** nach einem Funktionskopf drückt aus, dass die Funktion durch den folgenden geschlossenen *Ausdruck* definiert ist. Dieser Ausdruck wird *ausgewertet* und liefert den Funktionswert.

Programm 1
Funktionaler
Pseudocode: Enthält
der Suchraum das
gesuchte Element?

```

has (S, x) is
  if is_empty (S) then
    false
  elseif item (S) = x then
    true
  else
    has (reduced (S), x)
  end

```

Programm 2
Funktionaler
Pseudocode: Enthält
der Suchraum das
gesuchte Element?

```

has (S, x) is
  if is_empty (S) or else item (S) = x then
    not is_empty (S)
  else
    has (reduced (S), x)
  end

```

3

Imperativ rekursiv

Der dritte Schritt wechselt vom funktionalen ins imperative Programmierparadigma. Um Ausdrücke in Anweisungen einzubetten, bietet Tabelle 2 zwei Formen.

Tabelle 2
Funktionale und
imperative Notationen

Funktional	Imperativ
<i>Ausdruck</i>	return <i>Ausdruck</i>
	result := <i>Ausdruck</i>

Die erste Form, die **return**-Anweisung, ist als Sprunganweisung ein unstrukturiertes Element unter den Ablaufsteuerungskonstrukten. Sie findet sich z.B. in *Component Pascal* und den *C**-Sprachen. Die zweite Form ist die Zuweisung an eine implizit vereinbarte lokale Variable result für den Funktionswert. Sie ist kein unstrukturiertes Element und findet sich z.B. in *Eiffel*. Ob „**return**“ oder „result :=“ vor einem Ausdruck steht: Syntaktisch ist der Unterschied gering. Die folgenden Umformungen führen wir meist mit der **return**-Form durch; die result-Form benutzen wir vor allem, wenn wir result zum Formulieren von Nachbedingungen brauchen.

Die funktionalen Programme 1 und 2 erhalten in den Programmen 3 und 4 ihre erste imperative Form. Den Paradigmenwechsel erzielen kleine Syntaxänderungen. Das Schlüsselwort **do** nach einem Funktionskopf drückt aus, dass die Funktion durch die folgende *Anweisungsfolge* definiert ist. Das textuelle, statische Ende der Anweisungsfolge ist eindeutig durch das Schlüsselwort **end** markiert. Die Anweisungsfolge wird *ausgeführt*, eine **return**-Anweisung bedeutet ein dynamisches Ende und liefert einen Funktionswert durch Auswertung ihres Ausdrucks.

Programm 3

Imperativer

Pseudocode: Enthält der Suchraum das gesuchte Element?

```

has (S, x)
  do
    return
    if is_empty (S) then
      false
    elseif item (S) = x then
      true
    else
      has (reduced (S), x)
    end
  end
end

```

Programm 4

Imperativer

Pseudocode: Enthält der Suchraum das gesuchte Element?

```

has (S, x)
  do
    return
    if is_empty (S) or else item (S) = x then
      not is_empty (S)
    else
      has (reduced (S), x)
    end
  end
end

```

Der vierte Schritt besteht darin, eine Art Distributivregel anzuwenden: Die **returns** werden in die Zweige der bedingten Ausdrücke verteilt, wodurch diese in den Programmen 5 und 6 zu bedingten Anweisungen mutieren.

Programm 5

Imperativer

Pseudocode mit bedingter Anweisung: Enthält der Suchraum das gesuchte Element?

```

has (S, x)
  do
    if is_empty (S) then
      return false
    elseif item (S) = x then
      return true
    else
      return has (reduced (S), x)
    end
  end
end

```

Programm 6

Imperativer

Pseudocode mit bedingter Anweisung: Enthält der Suchraum das gesuchte Element?

```

has (S, x)
  do
    if is_empty (S) or else item (S) = x then
      return not is_empty (S)
    else
      return has (reduced (S), x)
    end
  end
end

```

Derekursivieren

Die Programme 5 und 6 enthalten imperativ formulierte endrekursive Funktionen. Sie lassen sich im fünften Schritt nach einem bekannten Umformungsschema (siehe [Skript Teil 1 Kapitel 3](#) S. 5) in die iterativen Rohvarianten der Programme 7 und 8 umformen.

4

Imperativ iterativ

Programm 7

Imperativer
Pseudocode
derekursiviert: Enthält
der Suchraum das
gesuchte Element?

```

has (S, x)
  local IS, lx
  do
    loop
      if is_empty (S) then
        return false
      elseif item (S) = x then
        return true
      else
        IS := reduced (S)
        lx := x
        return has (reduced (S), x)
        S := IS
        x := lx
      end
    end
  end
end

```

Programm 8

Imperativer
Pseudocode
derekursiviert: Enthält
der Suchraum das
gesuchte Element?

```

has (S, x)
  local IS, lx
  do
    loop
      if is_empty (S) or else item (S) = x then
        return not is_empty (S)
      else
        IS := reduced (S)
        lx := x
        return has (reduced (S), x)
        S := IS
        x := lx
      end
    end
  end
end

```

Der sechste Schritt vereinfacht in Programm 7 die eingefügten Zuweisungen und liefert Programm 9, dessen unbedingte Schleife zwei Ausgänge hat.

Programm 9

Imperativer
Pseudocode iterativ
unstrukturiert: Enthält
der Suchraum das
gesuchte Element?

```

has (S, x)
  do
    loop
      if is_empty (S) then
        return false
      elseif item (S) = x then
        return true
      else
        S := reduced (S)
      end
    end
  end
end

```

Die erste Abbruchbedingung lässt sich in eine Kopfbedingung einer kopfgesteuerten Bedingungsschleife wandeln:

Programm 10

Imperativer
Pseudocode iterativ
teilstrukturiert: Enthält
der Suchraum das
gesuchte Element?



```

has (S, x)
  do
    until is_empty (S) loop
      if item (S) = x then
        return true
      else
        S := reduced (S)
      end
    end
  end
  return false
end

```

Eine Bedingungsschleife mit einem zusätzlichem Ausgang aus dem Schleifenrumpf zu verquicken gefällt Strukturästheten nicht. Eine andere Umformung von Programm 9 belässt die unbedingte Schleife und verschleiert die Dreifachauswahl im Schleifenrumpf durch Streichen der beiden Zweigtrenner **else**. Übrig bleibt eine Folge von zwei einseitigen Auswahlanweisungen und einer Zuweisung in Programm 11:

Programm 11

Imperativer
Pseudocode iterativ
unstrukturiert mit
Wächtern: Enthält der
Suchraum das
gesuchte Element?



```

has (S, x)
  do
    loop
      if is_empty (S) then return false end
      if item (S) = x then return true end
      S := reduced (S)
    end
  end
end

```

Fowlers
Wächterbedingung

Die **if-return**-Bedingungen nennt Martin Fowler nach Kent Beck Wächterbedingungen (*guard clause*) [Fow99] S. 250–251, [Fow00] S. 255. An ihnen riecht ungut, dass der Schleier vor der Auswahl erst fällt, wenn die Anweisung bis zum **return** gelesen und der Schleifenausgang erkannt ist. Der Geruch bessert sich, wenn der Wächterzweig nicht wie üblich eingerückt ist, sondern die ganze Anweisung in einer Zeile steht. Wächter können vorteilhaft sein, wenn der **else**-Zweig lang und tief geschachtelt ist, weil er ohne **else** die Zweigschachtel verliert.

Parnas'
verallgemeinerte
Steueranweisung

Keines der Programme 10 und 11 variiert Programm 9 befriedigend. Trotzdem gibt es eine bessere Variante mit einer Steueranweisung, die David Lorge Parnas schon in den 1980er Jahren vorgeschlagen hat [Par83]. Betrachten wir die Struktur in Programm 9: Die Schleife enthält nur eine Mehrfachauswahl, aus deren Zweigen entweder die Funktion und damit die Schleife verlassen oder der Schleifenrumpf wiederholt wird. Diese oft vorkommende Situation abstrahiert die parnassche Steueranweisung: Sie vereinheitlicht und verallgemeinert Schleifen und Auswahlanweisungen so, dass sie allein zur Ablaufsteuerung genügt. Programm 12 ist die mit Parnas' verallgemeinerter Steueranweisung (aber anderer Syntax) formulierte Variante von Programm 9.

Programm 12

Parnasscher
Pseudocode mit dicker
Syntax: Enthält der
Suchraum das
gesuchte Element?

```

has (S, x)
  do
    when is_empty (S) then
      result := false exit
    elsewhen item (S) = x then
      result := true exit
    else
      S := reduced (S) cycle
    end
  end
end

```

Eingang

Die Schlüsselwörter **when**, **elsewhen**, **exit**, **cycle** sind von den üblichen Schlüsselwörtern **if**, **elseif**, **break**, **continue**, **repeat** verschieden gewählt, um Verwechslungen der Anweisungsarten zu vermeiden. Die Syntax und Semantik von **when-elsewhen-else-end**

ähnelt jedoch der von **if-elseif-else-end**: **elsewhen** kann beliebig oft vorkommen, **else** ist optional. Die Bedingungen nach **when** und **elsewhen** werden wie bei **if-elseif** der Reihe nach ausgewertet, bis eine Bedingung **true** liefert, dann wird der folgende **then**-Zweig ausgeführt, sonst der **else**-Zweig, sofern er da ist, sonst endet die **when**-Anweisung.

Ausgang

Der Unterschied zur üblichen Auswahlanweisung liegt am Ende der Zweige, wo explizit über den weiteren Ablauf entschieden wird: **exit** bedeutet, dass die **when**-Anweisung beendet wird, der Ablauf also nach **end** weitergeht; **cycle** bedeutet, dass die **when**-Anweisung wiederholt wird, der Ablauf also zu **when** zurückkehrt. **exit** und **cycle** ähneln **break** bzw. **continue** bei **C***-Schleifen. Es sind jetzt die einzigen Sprunganweisungen! **return** wird durch die implizit vereinbarte Variable **result** ersetzt, deren letzten Wert die Funktion automatisch zurückgibt.

Die in Programm 12 vorgestellte Syntaxvariante der parnasschen Steueranweisung braucht sieben Schlüsselwörter. Die „dicke“ Syntax kann abmagern: **else** ist durch **elsewhen true** ersetzbar, **elsewhen** durch den von der EBNF und der wirthschen CASE-Anweisung bekannten Oderstrich „|“, **then**, **exit** und **cycle** durch Pfeile in die entsprechenden Richtungen: $\rightarrow$, $\downarrow$, $\uparrow$. Schließlich ist **exit** ($\downarrow$) verzichtbar, indem der Sprung zu **end** zum impliziten Standard erklärt wird. Die „dünne“ Syntax mit zwei Schlüsselwörtern und drei Sonderzeichen verschlankt Programm 12 zu Programm 13, das Programm 11 ähnelt, aber statt drei Steueranweisungen nur eine benötigt. Programm 11 lässt sich als Implementation von Programm 13 mit konventionellen Steueranweisungen interpretieren.

Programm 13

Parnasscher
Pseudocode mit
dünner Syntax: Enthält
der Suchraum das
gesuchte Element?

```
has (S, x)
  do
    when
      | is_empty (S)  $\rightarrow$  result := false
      | item (S) = x  $\rightarrow$  result := true
      | true  $\rightarrow$  S := reduced (S)  $\uparrow$ 
    end
  end
end
```

Leider hat bisher kein Sprachentwerfer Parnas' generalisierte Steueranweisung aufgegriffen. Im Gegenteil, auch neue Sprachen bieten differenzierte Auswahl- und Wiederholungsanweisungen für Spezialfälle. Nach diesem Ausflug in die Welt des Sprachentwurfs vereinfachen wir auch in Programm 8 die Zuweisungen und transformieren die unbedingte Schleife in eine Bedingungschleife, woraus sich Programm 14 ergibt.

Programm 14

Imperativer
Pseudocode iterativ
strukturiert: Enthält der
Suchraum das
gesuchte Element?

```
has (S, x)
  do
    until is_empty (S) or else item (S) = x loop
      S := reduced (S)
    end
    return not is_empty (S)
  end
```

Die Programme 9, 10 und 11 enthalten je zwei **return**-Anweisungen, wovon mindestens eine im Schleifenrumpf steht und so statisch nicht auf das textuelle Ende der Funktion trifft und dynamisch das Verlassen der Funktion aus dem Schleifenrumpf heraus bewirkt. Deshalb sind diese Lösungen nicht strukturiert und in streng strukturierten Sprachen wie *Eiffel* nicht formulierbar. Dagegen enthält Programm 14 nur eine **return**-Anweisung am statischen Ende. Deshalb ist die Lösung strukturiert und auch in *Eiffel* formulierbar. Ihr Nachteil ist, dass die Bedingung **is_empty (S)** dupliziert vorkommt.

5

Objektorientiert

Der siebte Schritt bringt die Funktionen mit dem Suchraum als Parameter in eine objektorientierte Form. Der Suchraum mutiert in ein Objekt einer Klasse mit Abfragen und Aktionen, deren Zuordnung zu den Funktionen Tabelle 3 zeigt.

Tabelle 3
Funktionale und
objektorientierte
Notationen

Funktional mit Funktionen	Objektorientiert mit Diensten	
	Abfragen	Aktionen
Anfangswert von S		init
has (S, x)	has (x)	
is_empty (S)	is_empty	
item (S)	item	
reduced (S)		reduce

Paradigma wechseln

Der Paradigmenwechsel zeigt sich beim Suchraum. Im funktionalen Paradigma entsteht beim Suchen von x in S eine Folge

$$S \supseteq S' \supseteq \dots \supseteq S^{(i)} \supseteq \dots \supseteq S^{(n)}$$

schrumpfender Suchräume, die mit $S^{(n)} = \emptyset$ oder $a_{S^{(n)}} = x$ endet. In den funktionalen Programmen 1 und 2 entsprechen den $S^{(i)}$ verschiedene Parameterwerte der rekursiven Aufrufe (has (reduced (S))). In den imperativen Programmen 9 und 14 gibt es für alle $S^{(i)}$ nur eine Variable, der iterativ verschiedene Werte zugewiesen werden ($S := \text{reduced}(S)$). In den objektorientierten Programmen 15 und 16 entspricht allen $S^{(i)}$ ein Suchraumobjekt S , das als Empfängerobjekt mit der Aktion $S.\text{init}$ den Anfangszustand S und mit der Aktion $S.\text{reduce}$ den Folgezustand $S^{(i+1)}$ des aktuellen Zustands $S^{(i)}$ einnimmt.

Programm 15
Objektorientierter
Pseudocode
unstrukturiert: Enthält
der Suchraum das
gesuchte Element?

```

has (x)
do
  init
  loop
    if is_empty then
      return false
    elseif item = x then
      return true
    else
      reduce
    end
  end
end
end

```

Programm 16
Objektorientierter
Pseudocode
strukturiert: Enthält der
Suchraum das
gesuchte Element?

```

has (x)
do
  init
  until is_empty or else item = x loop
    reduce
  end
  return not is_empty
end
end

```

Typisieren

Die bisherigen Programmvarianten sind nicht typisiert. Der Suchraum erscheint in den Programmen 15 und 16 nicht explizit, ist aber implizit ein Objekt einer Klasse. Fassen wir Klassen als Typen auf, so stellt sich die Frage, ob nicht auch anderen Größen in den Programmen Typen zugeordnet sein sollen. Der achte Schritt präzisiert die Problemlösung durch Typisieren der Suchfunktion (Suchabfrage) has.

Programm 17

Oo Pseudocode
typisiert unstrukturiert:
Enthält der Suchraum
das gesuchte
Element?

```
has (x : ELEMENT) : BOOLEAN
do
  init
  loop
    if is_empty then
      return false
    elseif item = x then
      return true
    else
      reduce
    end
  end
end
end
```

Programm 18

Oo Pseudocode
typisiert strukturiert:
Enthält der Suchraum
das gesuchte
Element?

```
has (x : ELEMENT) : BOOLEAN
do
  init
  until is_empty or else item = x loop
    reduce
  end
  return not is_empty
end
```

Einbetten

Damit der Suchraum wieder explizit erscheint, werden die Programme 17 und 18 im neunten Schritt in den Kontext der Suchraumklasse SEARCHSPACE eingebettet. Zwischen öffentlichen (allgemein nutzbaren) und geschützten (nur von Erweiterungsklassen nutzbaren) Diensten der Klasse unterscheiden die C++ entlehnten Überschriften **public** bzw. **protected**. Bei den Diensten sind Abfragen, die einen Wert liefern, aber keinen Zustand ändern, von Aktionen, die einen Zustand ändern, aber nichts liefern, durch die Überschriften **queries** bzw. **actions** getrennt. Alle Dienste erscheinen typisiert. Das Ergebnis ist Programm 19 mit einem Platzhalter für die Suchabfrage.

Programm 19

Oo Pseudocode
typisiert: Suchraum

```
class SEARCHSPACE
public
  queries
    has (x : ELEMENT) : BOOLEAN
      -- body of program 17 or 18
    end
  protected
  queries
    is_empty : BOOLEAN
    item : ELEMENT
  actions
    init
    reduce
end
```

Generizität nutzen

Alle Objekte der Klasse SEARCHSPACE von Programm 19, etwa vereinbart durch `searchspace : SEARCHSPACE`, haben als Elementtyp denselben Typ ELEMENT, der zwar nicht spezifiziert ist und auch nicht spezifiziert sein muss, aber in SEARCHSPACE festgelegt ist. Wünschenswert sind Suchraumobjekte, deren Elementtypen nicht schon bei der Definition der Klasse, sondern erst bei den Vereinbarungen der Objekte festgelegt sind. Der zehnte Schritt erfüllt diesen Wunsch, indem er SEARCHSPACE in Programm 20 zu einer generischen Klasse verallgemeinert mit dem Elementtyp als formalem generischen Parameter, der E statt ELEMENT heißt, um Konflikte mit globalen Typnamen zu vermeiden, und mit **generic** markiert ist.

Programm 20

Oo Pseudocode

generisch: Suchraum

```

class SEARCHSPACE [generic E]
public
  queries
    has (x : E) : BOOLEAN
      -- body of program 17 or 18
    end
  protected
    queries
      is_empty : BOOLEAN
      item : E
    actions
      init
      reduce
  end

```

Mit Programm 20 wird bei einer Objektvereinbarung zu SEARCHSPACE für E ein konkreter Typ, etwa ELEMENT, eingesetzt:

```
searchspace : SEARCHSPACE [ELEMENT]
```

Abstrahieren
Partiell implementieren

Gammas
Entwurfsmuster
Schablonenmethode

Typfragen sind vorerst geklärt, wenden wir uns den Diensten zu. Die öffentliche Suchabfrage has ist mit den geschützten Diensten init, is_empty, item und reduce schon vollständig implementiert, es handelt sich um eine *Schablonenmethode (template method)*, die Anwendung eines Entwurfsmusters aus dem Katalog von Erich Gamma u.a. [GHJV94], [GHJV04]. Die benutzten Dienste sind abstrakt, eine Eigenschaft, die der elfte Schritt in der Klassendefinition in Programm 21 mit dem *Component Pascal* und *Java* entlehnten **abstract** ausdrückt.

Programm 21

Oo Pseudocode

abstrakt generisch:
Suchraum

```

abstract class SEARCHSPACE [generic E]
public
  queries
    has (x : E) : BOOLEAN
      -- Template method using services of protected part.
      -- body of program 17 or 18
    end
  protected
    queries
      is_empty : BOOLEAN
      abstract
      item : E
      abstract
    actions
      init
      abstract
      reduce
      abstract
  end

```

Mit Programm 21 sind syntaktische und statisch semantische Aspekte der Suchraumklasse spezifiziert: Welche Dienste gibt es? Welche Dienste sind Abfragen, welche Aktionen? Wie heißen sie? Welche Parameter und Ergebnisse haben sie? Welchen Typs? Welche Typen sind generisch, welche konkret? Welche Dienste sind öffentlich, welche geschützt? Welche Dienste sind abstrakt, welche implementiert?

6

Vertraglich spezifiziert

Vertraglich
spezifizieren

Meyers
Design by Contract

Offen ist die Spezifikation der dynamischen Semantik, des Verhaltens der Dienste. Im zwölften Schritt wenden wir Bertrand Meyers Methode *Spezifikation durch Vertrag* (*Design by Contract*) auf Programm 21 an [Mey09], [Hug01]. Als Verträge sind bei den Diensten Vor- und Nachbedingungen zu ergänzen, markiert mit **pre** bzw. **post**. Welche Dienste lassen sich so spezifizieren? `has`, `init` und `is_empty` stellen keine Vorbedingungen, ihre Ergebnisse bzw. Effekte lassen sich nicht mit Nachbedingungen ausdrücken. Aber zu `item` und `reduce` finden wir Vorbedingungen in der Rekursionsformel 3 S. 3 und erhalten damit Programm 22. Da das gesuchte Element `x` in der Vorbedingung von `reduce` vorkommt, ändern wir die Signatur von `reduce`: Es erhält einen Parameter, sodass sein Aufrufer `has` ihm `x` übergeben kann. Auch `init` kann von `x` abhängen! Bei `has` setzen wir exemplarisch den Rumpf von Programm 18 ein.

Programm 22

Oo Pseudocode
vertraglich spezifiziert,
partiell implementiert:
Suchraum

```

abstract class SEARCHSPACE [generic E]
public
  queries
    has (x : E) : BOOLEAN
      -- Template method using services of protected part.
    do
      init (x)
      until is_empty or else item = x loop
        reduce (x)
      end
      return not is_empty
    end
  protected
    queries
      is_empty : BOOLEAN
        abstract
      item : E
        pre
          not is_empty
        abstract
    actions
      init (x : E)
        abstract
      reduce (x : E)
        pre
          not is_empty and then item # x
        abstract
end

```

SEARCHSPACE in Programm 22 ist eine *Konzeptklasse*, die das Konzept Suchraum erfasst. Es ist keine Schnittstellenklasse, weil ihr Dienste zum Füllen des Suchraums mit Elementen fehlen. Das mit Programm 22 erreichte Stadium der vertraglichen Spezifikation ist nicht nur mit Spezifikationssprachen wie *Eiffel*, *OCL* und *Spec#* erreichbar, sondern auch mit Implementationssprachen wie *Component Pascal*, *C++*, *Java* und *C#*, sofern sie Zusicherungen bieten, um Vertragsbedingungen auszudrücken.

Man könnte geneigt sein, aus der rekursiven Definition des Suchraums, Eigenschaft (3), S. 2, die Bedingung

$$S' \subseteq S \setminus \{a_S\}$$

in die Nachbedingung

post **not** **has** (**old** **item**)

von `reduce` umzuformen, doch birgt dies drei Probleme.

Erstens muss sich der Aufruf von `has` auf den reduzierten Zustand des Suchraums in der Nachbedingung beziehen, aber `has` ruft zuerst `init` auf, welches das Suchraumobjekt in den Anfangszustand versetzt. Dies ist ein versteckter Nebeneffekt von `has`. Nebeneffekte dürfen in Verträgen nicht vorkommen, also darf `has` nicht in der Nachbedingung von `reduce` aufgerufen werden. Zweitens darf die Nachbedingung nicht zur endlosen Rekursion `has` $\rightarrow$ `reduce` $\rightarrow$ `has` $\rightarrow$... führen.

Drittens, vorausgesetzt die ersten beiden Probleme sind gelöst, ergibt sich ein Problem aus dem Unterschied zwischen dem mathematischen Konzept Menge und dem programmiersprachlichen Konzept Datenstruktur: In einer Menge ist jedes Element nur einmal enthalten, in einer Datenstruktur können Werte und Objekte mehrfach vorkommen. Entfernt man das aktuelle Element a_S aus S , ist es draußen. Reduziert man den Suchraum um `item`, so kann trotzdem sowohl der Wert von `item` (bei Wertsemantik) als auch das durch `item` bezeichnete Objekt (bei Bezugssemantik) noch im reduzierten Suchraum vorkommen!

Man könnte die semantische Lücke zwischen mathematischem Modell und Programm überbrücken, indem man den Begriff Suchraum mit Multimengen (die Elemente mehrfach enthalten können) statt Mengen definiert. Wir verzichten jedoch der Einfachheit halber darauf und achten statt dessen bei den folgenden Implementationen von `reduce` darauf, dass es als Nachbedingung garantiert:

„Im reduzierten Suchraum kommt der alte `item`-Bezug einmal weniger vor.“

7

Linear

Wir haben das spezielle Suchproblem (6) S. 2 und seine Lösung in Programm 22 auf hohem Abstraktionsniveau formalisiert, präzisiert und spezifiziert. Der diesem Stadium folgende Konkretisierungsschritt legt für die Elemente des Suchraums eine Struktur fest, eine konkrete Datenstruktur. Diese Datenstruktur ermöglicht es, die abstrakten Dienste zu implementieren.

Dieser Abschnitt befasst sich mit **linearer Suche**. Wir legen hier fest, dass

- (1) die Menge der Elemente im Suchraum durch eine Reihung implementiert ist;
- (2) die Reihung zum Suchen eines Elements von unten nach oben traversiert wird, wobei ein Index den Indexbereich der Reihung beginnend bei der Untergrenze durchläuft;
- (3) der aktuelle Suchraum aus den Reihungselementen mit Indizes vom aktuellen Index bis zur Obergrenze besteht;
- (4) das aktuelle Element das Reihungselement zum aktuellen Index ist;
- (5) der Suchraum reduziert wird, indem nur das aktuelle Element entfernt wird, was durch Erhöhen des Index geschieht.

Um diese verbale, verbose Konkretisierung zu formalisieren, sind einige Details der Umsetzung in eine Programmiersprache festzulegen. Dabei behalten wir das Abstraktionsniveau der Pseudosprache bei, die sich leicht in verschiedene Implementationssprachen übersetzen lässt.

- (6) Für Reihungen steht die generische Klasse `ARRAY [generic E]` bereit, `E` ist der Elementtyp. Der Indexbereich wird bei der Objektvereinbarung durch **range** `low .. high` mit der Invariante `low <= high + 1` spezifiziert.
- (7) Die Indexbereichsgrenzen eines mit `array` bezeichneten Reihungsobjekts sind durch `array.low` und `array.high` abfragbar.
- (8) Auf Reihungselemente wird mit der verbreiteten Klammernotation zugegriffen: `array [index]`.

Programm 23

Oo Pseudocode
implementierte
Abstraktion: Lineare
Suchreihung

```

class LINEAR_SEARCHARRAY [generic E] extends SEARCHSPACE [generic E]
protected
  data
    array : ARRAY [E] range low .. high
    index : INTEGER

  queries
    is_empty : BOOLEAN
      defined
        do
          return index > array.high
        end

    item : E
      pre
        -- not is_empty                -- from SEARCHSPACE
      defined
        do
          return array [index]
        end

  invariants
    array.low <= index
    index <= array.high + 1

  actions
    init (x : E)
      defined
        do
          index := array.low
        end

    reduce (x : E)
      pre
        -- not is_empty and then item # x  -- from SEARCHSPACE
      defined
        do
          index := index + 1
        end

end

```

Mit diesen Details realisieren wir die Konkretisierung in Programm 23 als konkrete Erweiterungsklasse `LINEAR_SEARCHARRAY` von `SEARCHSPACE` in Programm 22, syntaktisch durch das *Java* entlehnte **extends** angezeigt. Die schon implementierte Suchabfrage `has` wird geerbt und deshalb in Programm 23 nicht wiederholt. Die Dienste `is_empty`, `item`, `init`, `reduce` werden mit den unter der Überschrift **data** vereinbarten konkreten Daten `array` und `index` implementiert. Das Schlüsselwort **defined** zeigt an, dass es sich um eine Definition eines abstrakten Vorgängers handelt.

Die Dienste erben die Verträge ihrer abstrakten Vorgänger. Die Verträge werden nur hier als Kommentar wiederholt, damit der Leser die Korrektheit der Implementation leicht (ohne Umblättern) prüfen kann. Als neue Vertragselemente erscheinen unter der Überschrift **invariants** zwei Implementationsinvarianten, die die Konsistenz der konkreten Datenstruktur ausdrücken: Der Index `index` darf die Untergrenze des Indexbereichs der Reihung `array` nicht unterschreiten und ihre Obergrenze um höchstens 1 überschreiten.

Was haben wir erreicht? Wir haben das lineare Suchproblem in fünf Dienste und zwei Klassen zerlegt: in eine wiederverwendbare Konzeptklasse und eine konkrete Klasse, die das Konzept implementiert; in eine Schablonenmethode, die die algorithmische Struktur der Problemlösung erfasst, und vier Dienste, die konzeptuell unwesentliche Details nach außen abstrahieren und innen konkretisieren. Um zu prüfen, ob das Ergeb-

nis korrekt ist, machen wir die Zerlegung rückgängig. Im ersten Schritt verflachen wir die beiden Klassen zu einer Klasse und erhalten Programm 24.

Programm 24
Oo Pseudocode
verflacht: Lineare
Suchreihung

```

class LINEAR_SEARCHARRAY [generic E]
public
  queries
    has (x : E) : BOOLEAN
    do
      init (x)
      until is_empty or else item = x loop
        reduce (x)
      end
      return not is_empty
    end
  protected
  data
    array : ARRAY [E] range low .. high
    index : INTEGER
  queries
    is_empty : BOOLEAN
    do
      return index > array.high
    end
    item : E
    pre
      not is_empty
    do
      return array [index]
    end
  invariants
    array.low <= index
    index <= array.high + 1
  actions
    init (x : E)
    do
      index := array.low
    end
    reduce (x : E)
    pre
      not is_empty and then item # x
    do
      index := index + 1
    end
end

```

Im zweiten Schritt expandieren wir die Aufrufe der untergeordneten Dienste in has und erhalten Programm 25. Die Vorbedingung von reduce wird zu einer Zusicherung (**assert**). Die Vorbedingung von item kann nicht als Anweisung in einem booleschen Ausdruck untergebracht werden.

Programm 25

Oo Pseudocode
 expandiert: Lineare
 Suchreihung

```

class LINEAR_SEARCHARRAY [generic E]
public
  queries
    has (x : E) : BOOLEAN
    do
      index := array.low
      until index > array.high or else array [index] = x loop
        assert index <= array.high and then array [index] # x
        index := index + 1
      end
      return index <= array.high
    end
protected
  data
    array : ARRAY [E] range low .. high
    index : INTEGER
  invariants
    array.low <= index
    index <= array.high + 1
end

```

In Programm 25 ist die Suchabfrage `has` als einzige Funktion übrig geblieben. Sie initialisiert `index` bei jedem Aufruf. Deshalb muss `index` nicht mehr global, sondern kann lokal in `has` vereinbart werden (**local**). (Ein lokaler Index könnte kurz mit `i` statt `index` benannt sein.) Die Klasseninvarianten zum Wertebereich von `index` werden zu Schleifeninvarianten (**loopinvariants**). Das Ergebnis dieser Umformung zeigt Programm 26.

Programm 26

Oo Pseudocode
 lokalisiert: Lineare
 Suchreihung

```

class LINEAR_SEARCHARRAY [generic E]
public
  queries
    has (x : E) : BOOLEAN
    local
      index : INTEGER
    do
      index := array.low
      loopinvariants
        array.low <= index
        index <= array.high + 1
      until index > array.high or else array [index] = x loop
        assert index <= array.high and then array [index] # x
        index := index + 1
      end
      return index <= array.high
    end
protected
  data
    array : ARRAY [E] range low .. high
  invariants
    array.low <= array.high + 1
end

```

In Programm 26 existiert neben `has` nur noch die Reihung `array` als globale Datenstruktur. `has` kann nur in dem globalen `array` suchen. Indem wir das globale `array` in einen Eingabeparameter von `has` wandeln, machen wir `has` flexibler. Die Klasseninvariante zu `array` wird zu einer Vorbedingung von `has`. Nachdem wir uns in Programm 26 davon überzeugt haben, dass die Anweisungen von `has` den Zusicherungen (**loopinvariants**,

assert) entsprechen, können wir diese Zusicherungen entfernen. Das Ergebnis dieser Umformungen zeigt Programm 27.

Programm 27
Oo Pseudocode
parametrisiert: Lineare
Suchreihung

```
class LINEAR_SEARCHARRAY [generic E]
public
  queries
    has (array : ARRAY [E]; x : E) : BOOLEAN
      pre
        array.low <= array.high + 1
      local
        index : INTEGER
      do
        index := array.low
        until index > array.high or else array [index] = x loop
          index := index + 1
        end
      return index <= array.high
    end
  end
end
```

In Programm 27 hat nur `has` den Schrumpfungsprozess überstanden. Die Klasse `LINEAR_SEARCHARRAY` hat mit ihren Daten auch ihre Existenzberechtigung verloren, denn wozu sollten verschiedene Objekte nützen, die keinen Zustand haben, sich also in nichts unterscheiden? Die Lösung ist nicht wirklich objektorientiert, sondern prozedural mit einer einzigen Funktionsprozedur, die allerdings einen generischen Parameter hat. Wir passen die Pseudosprache an diese Situation an und erhalten Programm 28.

Programm 28
Prozeduraler
Pseudocode mit
generischer Funktion:
Lineare Suchreihung

```
has [generic E] (array : ARRAY [E]; x : E) : BOOLEAN
  local
    index : INTEGER
  do
    index := array.low
    until index > array.high or else array [index] = x loop
      index := index + 1
    end
  return index <= array.high
end
```

Programm 28 mit Programm 14 verglichen zeigt, dass die Rückführung der Lösung vom objektorientierten in den prozeduralen Programmierstil die algorithmische Struktur erhalten hat. Zudem enthält Programm 28 die **strukturierte Standardform des linearen Suchalgorithmus**. Hätten wir in Programm 22 für `has` den Rumpf von Programm 9, 10 oder 11 statt von Programm 14 eingesetzt, so stünde in Programm 28 eine unstrukturierte Variante des linearen Suchalgorithmus. (Der Leser möge diese alternativen Umformungen selbst ausführen.)

Aufgabe 1
Lineare Suche in
verketteter Liste

Führen Sie die Schritte dieses Abschnitts zur linearen Suche mit einer einfach verketteten Liste als konkrete Datenstruktur durch!

8

Binär

Dieser Abschnitt wiederholt das Vorgehen des vorigen Abschnitts mit binärer Suche. Bei linearer Suche lässt sich der Suchraum bei jeder Iteration nur um das aktuelle Element reduzieren. Deshalb ist der Laufzeitaufwand (*run-time complexity*) der linearen Suche durch die Anzahl n der Elemente linear beschränkt:

$$RTC(\text{linear search}) \in O(n).$$

Das ist bei großem n nachteilig. Je stärker der Suchraum bei jedem Schritt reduziert wird, desto niedriger ist der Laufzeitaufwand. Bestenfalls wird der Suchraum bei jedem Schritt halbiert, dann ist er nach $\lg(n)$ Schritten einelementig. Eine Hälfte des Suchraums lässt sich aber nur entfernen, wenn gesichert ist, dass das gesuchte Element nicht in dieser Hälfte vorkommt. Das trifft zu, wenn der Suchraum sortiert und als aktuelles Element das mittlere Element gewählt ist. Das Sortiertsein setzt eine Vollordnung auf den Elementen voraus. Dieses Verfahren heißt **binäre Suche**. Sein Laufzeitaufwand ist logarithmisch beschränkt:

$$RTC(\text{binary search}) \in O(\log(n)).$$

Das ist sehr vorteilhaft! Wir legen hier fest, dass

- (1) auf den Elementen des Suchraums eine Vollordnung definiert ist;
- (2) die Menge der Elemente im Suchraum durch eine Reihung implementiert ist;
- (3) die Reihung gemäß der Vollordnung der Elemente (aufsteigend) sortiert ist;
- (4) in der Reihung zum Suchen eines Elements von der Mitte in die linke oder rechte Hälfte gesprungen wird, wobei sich ein mittlerer Index zwischen einem linken und einem rechten Index bewegt;
- (5) der aktuelle Suchraum aus den Reihungselementen mit Indizes vom linken bis zum rechten Index besteht;
- (6) das aktuelle Element das Reihungselement zum mittleren Index ist;
- (7) der Suchraum reduziert wird, indem mit dem aktuellen Element auch die Hälfte (der kleineren Elemente) links oder (der größeren Elemente) rechts davon entfernt wird;
- (8) der Suchraum solange halbiert wird, bis er leer oder das gesuchte Element gefunden ist.

Um diese verbale, verbose Konkretisierung des **einfachen binären Suchalgorithmus** zu formalisieren, legen wir dieselben Details wie in Abschnitt 7 fest, nämlich (6) bis (8) S. 13. Damit realisieren wir die Konkretisierung in Programm 29 als zweite konkrete Erweiterungsklasse `BINARY_SEARCHARRAY` von `SEARCHSPACE` in Programm 22. `COMPARABLE` ist eine Konzeptklasse, die das Konzept Vollordnung mit den Relationen $<$, $>$, $\leq$, $\geq$ erfasst. Die Dienste `is_empty`, `item`, `init`, `reduce` werden hier mittels der konkreten Daten `array`, `left`, `mid`, `right` implementiert. Als neue Vertragselemente erscheinen vier Implementationsinvarianten, die die Konsistenz der konkreten Datenstruktur ausdrücken. Für die Invariante, dass die Reihung sortiert ist, ist die Funktion `sorted` definiert, die mit einer Variation des strukturierten linearen Suchalgorithmus implementiert ist.

Programm 29

Oo Pseudocode
implementierte
Abstraktion: Binäre
Suchreihung

```

class BINARY_SEARCHARRAY [generic E extends COMPARABLE]
  extends SEARCHSPACE [generic E]
protected
  data
    array : ARRAY [E] range low .. high
    left, mid, right : INTEGER
  queries
    is_empty : BOOLEAN
    defined
    do
      return left > right
    end
    item : E
    pre
      -- not is_empty                -- from SEARCHSPACE
    defined
    do
      return array [mid]
    end
    sorted (a : ARRAY [E]) : BOOLEAN
    local
      i : INTEGER
    do
      i := a.low
      until i >= a.high or else a [i] > a [i + 1] loop
        i := i + 1
      end
      return i >= a.high
    end
  invariants
    array.low <= array.high + 1
    sorted (array)
    array.low <= left
    right <= array.high
    left <= right implies (left <= mid and mid <= right)
  actions
    init (x : E)
    defined
    do
      left := array.low
      right := array.high
      mid := (left + right) div 2
    end
    reduce (x : E)
    pre
      -- not is_empty and then item # x  -- from SEARCHSPACE
    defined
    do
      if item < x then
        left := mid + 1
      else
        right := mid - 1
      end
      mid := (left + right) div 2
    end
  end
end

```

Um Programm 29 zu verifizieren, verflachen wir die beiden Klassen zu einer Klasse und erhalten Programm 30.

Programm 30

Oo Pseudocode

verflacht: Binäre

Suchreihung

```

class BINARY_SEARCHARRAY [generic E extends COMPARABLE]
public
  queries
    has (x : E) : BOOLEAN
    do
      init (x)
      until is_empty or else item = x loop
        reduce (x)
      end
      return not is_empty
    end
protected
  data
    array : ARRAY [E] range low .. high
    left, mid, right : INTEGER
  queries
    is_empty : BOOLEAN
    do
      return left > right
    end
    item : E
    pre
      not is_empty
    do
      return array [mid]
    end
    sorted (a : ARRAY [E]) : BOOLEAN
      -- body of program 29
    end
  invariants
    array.low <= array.high + 1
    sorted (array)
    array.low <= left
    right <= array.high
    left <= right implies (left <= mid and mid <= right)
  actions
    init (x : E)
    do
      left := array.low
      right := array.high
      mid := (left + right) div 2
    end
    reduce (x : E)
    pre
      not is_empty and then item # x
    do
      if item < x then
        left := mid + 1
      else
        right := mid - 1
      end
      mid := (left + right) div 2
    end
end

```

Im zweiten Schritt expandieren wir die Aufrufe der untergeordneten Dienste in has und erhalten Programm 31.

Programm 31

Oo Pseudocode
 expandiert: Binäre
 Suchreihung

```

class BINARY_SEARCHARRAY [generic E extends COMPARABLE]
public
  queries
    has (x : E) : BOOLEAN
      do
        left := array.low
        right := array.high
        mid := (left + right) div 2
        until left > right or else array [mid] = x loop
          assert left <= right and then array [mid] # x
          if array [mid] < x then
            left := mid + 1
          else
            right := mid - 1          -- array [mid] > x
          end
          mid := (left + right) div 2
        end
        return left <= right
      end
protected
  data
    array : ARRAY [E] range low .. high
    left, mid, right : INTEGER
  queries
    sorted (a : ARRAY [E]) : BOOLEAN
      -- body of program 29
    end
  invariants
    array.low <= array.high + 1
    sorted (array)
    array.low <= left
    right <= array.high
    left <= right implies (left <= mid and mid <= right)
end

```

In Programm 31 sind die Suchabfrage `has` und die Sortiertprüffunktion `sorted` als einzige Funktionen übrig geblieben. `sorted` ist unabhängig von den globalen Daten. `has` initialisiert `left`, `mid`, `right` bei jedem Aufruf. Deshalb müssen diese Daten nicht mehr global, sondern können lokal in `has` vereinbart werden. Die Klasseninvarianten zu ihren Wertebereichen werden zu Schleifeninvarianten. Das Ergebnis dieser Umformung zeigt Programm 32.

Programm 32

Oo Pseudocode

lokalisiert: Binäre

Suchreihung

```

class BINARY_SEARCHARRAY [generic E extends COMPARABLE]
public
  queries
    has (x : E) : BOOLEAN
      local
        left, mid, right : INTEGER
      do
        left := array.low
        right := array.high
        mid := (left + right) div 2
        loopinvariants
          array.low <= left
          right <= array.high
          left <= right implies (left <= mid and mid <= right)
        until left > right or else array [mid] = x loop
          assert left <= right
          if array [mid] < x then
            left := mid + 1
          else
            right := mid - 1
          end
          mid := (left + right) div 2
        end
        return left <= right
      end
    end
  protected
    data
      array : ARRAY [E] range low .. high
    queries
      sorted (a : ARRAY [E]) : BOOLEAN
        -- body of program 29
      end
    invariants
      array.low <= array.high + 1
      sorted (array)
    end

```

In Programm 32 existiert neben `has` und `sorted` nur noch die Reihung `array` als globale Datenstruktur. Wir wandeln das globale `array` in einen Eingabeparameter von `has`. Die Klasseninvarianten zu den Indexgrenzen und der Sortiertheit der Reihung werden zu Vorbedingungen von `has`. Die Abfrage `sorted` muss öffentlich werden, damit Kunden von `has` dessen Vorbedingung prüfen können. Nachdem wir uns in Programm 32 davon überzeugt haben, dass die Anweisungen von `has` den Zusicherungen (**loopinvariants**, **assert**) entsprechen, können wir diese Zusicherungen entfernen. Das Ergebnis dieser Umformungen zeigt Programm 33.

Programm 33

Oo Pseudocode
parametrisiert: Binäre
Suchreihung

```

class BINARY_SEARCHARRAY [generic E extends COMPARABLE]
public
  queries
    has (array : ARRAY [E]; x : E) : BOOLEAN
      pre
        array.low <= array.high + 1
        sorted (array)
      local
        left, mid, right : INTEGER
      do
        left := array.low
        right := array.high
        mid := (left + right) div 2
        until left > right or else array [mid] = x loop
          if array [mid] < x then
            left := mid + 1
          else
            right := mid - 1
          end
          mid := (left + right) div 2
        end
        return left <= right
      end
    sorted (a : ARRAY [E]) : BOOLEAN
      -- body of program 29
    end
end

```

In Programm 33 haben nur `has` und `sorted` den Schrumpfungsprozess überstanden. Da die Klasse `BINARY_SEARCHARRAY` mit ihren Daten auch ihre Existenzberechtigung verloren hat, eliminieren wir sie und schreiben die Lösung in Programm 34 im prozeduralen Stil hin. Die Korrektheit des einfachen binären Suchalgorithmus in Programm 34 lässt sich wegen seiner Symmetrie bezüglich `left` und `right` leicht auch ohne Zusicherungen und formalen Beweis durch genaues Betrachten prüfen.

Programm 34

Prozeduraler
Pseudocode mit
generischen
Funktionen: Binäre
Suchreihung

```

has [generic E extends COMPARABLE] (array : ARRAY [E]; x : E) : BOOLEAN
pre
  array.low <= array.high + 1
  sorted (array)
local
  left, mid, right : INTEGER
do
  left := array.low
  right := array.high
  mid := (left + right) div 2
  until left > right or else array [mid] = x loop
    if array [mid] < x then
      left := mid + 1
    else
      right := mid - 1
    end
    mid := (left + right) div 2
  end
  return left <= right
end
sorted [generic E extends COMPARABLE] (a : ARRAY [E]) : BOOLEAN
-- body of program 29
end

```

Ein Vergleich von Programm 34 mit Programm 14 zeigt, dass beim Rückführen der Lösung vom objektorientierten in den prozeduralen Programmierstil die algorithmische Struktur erhalten bleibt. Vermeintlich nachteilig an Programm 34 ist, dass statisch die Zuweisung an `mid` doppelt vorkommt und dynamisch der Wert für `mid` zu früh berechnet wird, bevor geprüft ist, ob er gebraucht wird. `mid` wird also einmal umsonst berechnet. Hätten wir das bei der Konkretisierung der Abstraktion vermeiden können? Im abstrakten Programm 22 für die Schablonenmethode `has` den unstrukturierten Rumpf von Programm 17 einsetzen, im konkreten Programm 29 die Zuweisung

```
mid := (left + right) div 2
```

in `init` streichen und vom Ende in `reduce` woandershin schieben – all das genügt nicht. (Der Leser möge dies selbst prüfen.) Es bedarf in Programm 22 einer weiteren Aktion `prepare`, die standardmäßig leer ist und in `has` von Programm 17 zwischen `is_empty` und `item` aufzurufen ist:

Programm 35

Oo Pseudocode mit zusätzlicher Aktion: Enthält der Suchraum das gesuchte Element?

```
has (x : ELEMENT) : BOOLEAN
do
  init
  loop
    if is_empty then
      return false
    else
      prepare                                -- establish additional preconditions for item
      if item = x then
        return true
      else
        reduce
      end
    end
  end
end
end
```

In Programm 29 ist `prepare` mit der Zuweisung an `mid` zu implementieren. Fazit: Algorithmenfeilerei zwecks marginaler Effizienzgewinne kann Abstraktion zwecks Wiederverwendbarkeit nicht verhindern, aber die Ästhetik der Lösung mindern.

Aufgabe 2

Suche in geordnetem Binärbaum

Führen Sie die Schritte dieses Abschnitts zur binären Suche mit einem geordneten Binärbaum (binären Suchbaum) als konkrete Datenstruktur durch!

9

Beschleunigt

Wer den Laufzeitaufwand RTC verringern will, der sollte weniger winzige Abzüge ($RTC - \delta$), mehr prozentuale Rabatte ($RTC * (1 - \delta)$) suchen. Rund 20 % Rabatt sind erreichbar, wenn in der Abbruchbedingung in Programm 34

```
left > right or else array [mid] = x
```

der zweite Operand gespart, der erste zu

```
left >= right
```

abgeschwächt, und erst nach der Schleife geprüft wird, ob `x` gefunden wurde. Diese und ergänzende Änderungen führen zum **beschleunigten binären Suchalgorithmus** in Programm 36. Er iteriert zwar im Mittel einmal mehr als der einfache Algorithmus, aber ab einer gewissen Größe des Suchraums übersteigt die Summe der Gewinne bei den Iterationen die konstanten Zusatzkosten.

Funktioniert der beschleunigte binäre Suchalgorithmus in Programm 36 wirklich wie gewünscht? Wegen seiner Asymmetrie bezüglich `left` und `right` ist seine Korrektheit nicht offensichtlich. Um sie prüfbar zu machen, versehen wir den Algorithmus in Pro-

Programm 37 mit Zusicherungen, insbesondere Schleifeninvarianten und einer Schleifenvariante, sowie mit erläuternden Kommentaren.

Programm 36

Prozeduraler
Pseudocode
beschleunigt: Binäre
Suchreihung

```
has [generic E extends COMPARABLE] (array : ARRAY [E]; x : E) : BOOLEAN
  pre
    array.low <= array.high + 1
    sorted (array)
  local
    left, mid, right : INTEGER
  do
    left := array.low
    right := array.high + 1
    until left >= right loop
      mid := (left + right) div 2
      if array [mid] < x then
        left := mid + 1
      else
        right := mid
      end
    end
  end
  return left <= array.high and then array [left] = x
end
```

Programm 37

Prozeduraler
Pseudocode
beschleunigt mit
Zusicherungen: Binäre
Suchreihung

```
has [generic E extends COMPARABLE] (array : ARRAY [E]; x : E) : BOOLEAN
  pre
    array.low <= array.high + 1
    sorted (array)
  local
    left, mid, right : INTEGER
  do
    left := array.low
    right := array.high + 1
    loopinvariants
      array.low <= left
      left <= right
      right <= array.high + 1
      for all k = array.low .. left - 1 : array [k] < x
      for all k = right .. array.high : array [k] >= x
    until left >= right loop
      assert left < right
      mid := (left + right) div 2
      assert left <= mid and mid < right
      if array [mid] < x then
        left := mid + 1
        -- exclude low .. mid
      else
        right := mid
        -- array [mid] >= x
        -- exclude mid + 1 .. high
      end
    loopvariant
      right - left
    end
    assert left = right
    assert for all k = array.low .. left - 1 : array [k] < x
    assert for all k = left .. array.high : array [k] >= x
    return left <= array.high and then array [left] = x
  end
```

Was ist zum Beweis der Korrektheit der Funktion zu zeigen? Erstens, dafür, dass die Schleife das Gewünschte leistet: Dass die Schleifeninvarianten vor und nach jeder Iteration gelten und mit der Abbruchbedingung die Schleifennachbedingungen ergeben. Zweitens, dafür, dass die Schleife terminiert: Dass die Schleifenvariante stets nicht-negativ bleibt und bei jeder Iteration kleiner wird. Drittens, dafür, dass das Funktions-

ergebnis korrekt ist: Dass der **return**-Ausdruck korrekt ist. Der Beweis der Schleifeninvarianten wird mit vollständiger Induktion geführt.

Verifikation

Beweis der Schleifeninvarianten. *Induktionsanfang:* Zu zeigen ist, dass die Schleifeninvarianten vor der ersten Iteration gelten. Aus der Reihungsinvarianten $\text{array.low} \leq \text{array.high} + 1$ in (6) S. 13, die in Programm 37 als Vorbedingung erscheint, und den Initialisierungen $\text{left} := \text{array.low}$ und $\text{right} := \text{array.high} + 1$ folgt $\text{array.low} \leq \text{left} \leq \text{right} \leq \text{array.high} + 1$. Nach der Initialisierung von left ist das Intervall $\text{array.low} .. \text{left} - 1$ leer, sodass die Formel **for all** $k = \text{array.low} .. \text{left} - 1 : \text{array}[k] < x$ trivialerweise gilt. Nach der Initialisierung von right ist $\text{right} .. \text{array.high} + 1$ leer, sodass **for all** $k = \text{right} .. \text{array.high} + 1 : \text{array}[k] \geq x$ trivialerweise gilt.

Induktionsschritt: Unter der *Induktionsvoraussetzung*, dass die Schleifeninvarianten vor dem Schleifenrumpf gelten, ist im *Induktionsschluss* zu zeigen, dass sie auch nach dem Schleifenrumpf gelten. Vorausgesetzt, $\text{array.low} \leq \text{left} \leq \text{right} \leq \text{array.high} + 1$ gilt vor dem Schleifenrumpf. Dann gilt wegen der Abbruchbedingung $\text{left} \geq \text{right}$ sogar $\text{left} < \text{right}$. Danach wird die Mitte mid zur nächstkleineren Ganzzahl abgerundet, sodass $\text{left} \leq \text{mid} < \text{right}$ gilt. Dann wird entweder left erhöht oder right erniedrigt, sodass danach entweder $\text{left} = \text{mid} + 1 \leq \text{right}$ oder $\text{left} \leq \text{mid} = \text{right}$, und in jedem Fall $\text{array.low} \leq \text{left}$ und $\text{right} \leq \text{array.high} + 1$ gilt.

Vorausgesetzt, **for all** $k = \text{array.low} .. \text{left} - 1 : \text{array}[k] < x$ gilt vor dem Schleifenrumpf. Falls left nicht erhöht wurde, gilt die Formel danach immer noch. Falls left auf $\text{mid} + 1$ erhöht wurde, gilt $\text{array}[\text{mid}] < x$. Weil array sortiert ist und $\text{mid} = \text{left} - 1$, gilt **for all** $k = \text{array.low} .. \text{mid} : \text{array}[k] < x \equiv \text{for all } k = \text{array.low} .. \text{left} - 1 : \text{array}[k] < x$.

Vorausgesetzt, **for all** $k = \text{right} .. \text{array.high} : \text{array}[k] \geq x$ gilt vor dem Schleifenrumpf. Falls right nicht erniedrigt wurde, gilt die Formel danach immer noch. Falls right auf mid erniedrigt wurde, gilt $\text{array}[\text{mid}] \geq x$. Weil array sortiert ist und $\text{mid} = \text{right}$, gilt **for all** $k = \text{mid} .. \text{array.high} : \text{array}[k] \geq x \equiv \text{for all } k = \text{right} .. \text{array.high} : \text{array}[k] \geq x$.

Beweis der Schleifenvarianten. Wegen der Schleifeninvariante $\text{left} \leq \text{right}$ ist die Schleifenvariante $\text{right} - \text{left} \geq 0$. Da entweder left erhöht oder right erniedrigt wird, wird die Differenz bei jeder Iteration kleiner.

Beweis der Schleifennachbedingungen. Aus der zweiten Schleifeninvariante $\text{left} \leq \text{right}$ und der Abbruchbedingung $\text{left} \geq \text{right}$ folgt die erste Nachbedingung $\text{left} = \text{right}$. Die zweite Nachbedingung ist genau die vierte Schleifeninvariante. Aus der fünften Schleifeninvarianten **for all** $k = \text{right} .. \text{array.high} : \text{array}[k] \geq x$ und $\text{left} = \text{right}$ folgt die dritte Nachbedingung **for all** $k = \text{left} .. \text{array.high} : \text{array}[k] \geq x$. Die Nachbedingungen bedeuten: Ist left ein gültiger Index, so ist es der kleinste Index, bei dem das gesuchte Element x steht, falls es überhaupt in der Reihung vorkommt. Ist left kein gültiger Index, so sind alle Elemente in der Reihung kleiner als das gesuchte.

Beweis der Korrektheit des Funktionsergebnisses. Falls $\text{left} = \text{array.high} + 1$, wurde das gesuchte Element nicht gefunden, weil entweder der initiale Suchraum leer ist ($\text{array.low} = \text{left} = \text{array.high} + 1$) oder das gesuchte Element größer als das oberste Element der Reihung ist ($\text{array}[\text{array.high}] < x$), sodass left über array.high hinaus erhöht wurde und right den Initialwert $\text{array.high} + 1$ behalten hat. Falls $\text{array.low} \leq \text{left} \leq \text{array.high} < \text{array.high} + 1$, bleibt zu prüfen, ob das Element beim Index left das gesuchte ist. $\square$

Konstruktion

Die hier exemplarisch angewandte Methode der Verifikation von Programmen ist analytisch: Zuerst ist das fertige Programm da, dann wird seine Korrektheit bewiesen. Aber wie entsteht das Programm? Die komplementäre, synthetische Methode der Programmkonstruktion beginnt mit einzelnen Programmelementen und kombiniert sie systematisch zu einem Programm. Bei der beschleunigten binären Suche starten wir bei der Fallunterscheidung im Schleifenrumpf und einer Schleifeninvariante und konstruieren

ieren daraus die Fallzweige, die Initialisierung vor und die Restarbeit nach der Schleife, wobei gelegentlich Entwurfsentscheidungen zu treffen sind.

Wegen ihrer Asymmetrie gibt es die beschleunigte binäre Suche in zwei Varianten. Bei beiden Varianten ist das Suchintervall $\text{left} \dots \text{right}$ stets nichtleer. Vor der Schleife gilt $\text{left} \leq \text{right}$, danach $\text{left} = \text{right}$. Falls das gesuchte Element gefunden wird, steht es bei den Indizes left und right .

Die obige Variante schlägt den Fall $\text{array}[\text{mid}] = x$ durch die Unterscheidung ($\text{array}[\text{mid}] < x$, $\text{array}[\text{mid}] \geq x$) der linken Hälfte des Suchintervalls zu.

- Falls $x \leq \text{array}[\text{mid}]$, geht die Suche in der linken Hälfte weiter. Da $x = \text{array}[\text{mid}]$ möglich ist, darf mid *nicht* aus der linken Hälfte entfernt werden, right darf *nicht* auf $\text{mid} - 1$ erniedrigt werden, sondern nur auf mid . Also wird right nie kleiner als mid und left . Ist x kleiner als jedes Element in array , oder ist x gleich dem kleinsten Element in array , so behält left seinen Anfangswert und right erreicht schließlich left . Ist x das kleinste Element in array , muss nach der Schleife $\text{array}.\text{low} = \text{left} = \text{right}$ gelten. Also *muss* $\text{array}.\text{low}$ der Anfangswert von left sein.
- Falls $\text{array}[\text{mid}] < x$, geht die Suche in der rechten Hälfte weiter, mid darf aus der rechten Hälfte entfernt werden, left wird auf $\text{mid} + 1$ erhöht, wird also größer als mid . Ist x größer als jedes Element in array , so behält right seinen Anfangswert und left erreicht schließlich right . Angenommen, der Anfangswert von right ist $\text{array}.\text{high}$. Da die Lösung allgemeingültig sein soll, sind leere Reihungen wie in vielen Programmiersprachen erlaubt. Bei leerer Reihung gilt $\text{array}.\text{high} = \text{array}.\text{low} - 1$. Damit gilt: Anfangswert von $\text{right} = \text{array}.\text{low} - 1 < \text{array}.\text{low} = \text{Anfangswert von left}$. Das widerspricht der Schleifeninvariante $\text{left} \leq \text{right}$. Also darf der Anfangswert von right nicht $\text{array}.\text{high}$ sein, es *muss* der nächstgrößere Wert $\text{array}.\text{high} + 1$ sein, der kein gültiger Reihungsindex ist. Ist also x größer als jedes Element in array , so überschreitet left die Obergrenze des Reihungsindexbereichs. Zusammengefasst gilt: Ist die Reihung leer oder das gesuchte Element größer als das größte in der Reihung, so haben left und right nach der Schleife den ungültigen Indexwert $\text{array}.\text{high} + 1$. Deshalb muss nach der Schleife zuerst mit $\text{left} \leq \text{array}.\text{high}$ die Gültigkeit des Index left geprüft werden, bevor geprüft wird, ob das gesuchte Element in $\text{array}[\text{left}]$ steht.

Die alternative Variante schlägt den Fall $\text{array}[\text{mid}] = x$ durch die Unterscheidung ($\text{array}[\text{mid}] \leq x$, $\text{array}[\text{mid}] > x$) der rechten Hälfte des Suchintervalls zu. Gegenüber obigen Ausführungen tauschen left und right ihre Rollen.

- Falls $\text{array}[\text{mid}] \leq x$, geht die Suche in der rechten Hälfte weiter. Da $x = \text{array}[\text{mid}]$ möglich ist, darf left nur auf mid erhöht werden, wird also nie größer als mid und right . Ist x größergleich jedem Element in array , so behält right seinen Anfangswert und left erreicht schließlich right . Ist x das größte Element in array , muss nach der Schleife $\text{left} = \text{right} = \text{array}.\text{high}$ gelten. Also *muss* $\text{array}.\text{high}$ der Anfangswert von right sein.
- Falls $x < \text{array}[\text{mid}]$, geht die Suche in der linken Hälfte weiter, right wird auf $\text{mid} - 1$ erniedrigt, wird also kleiner als mid . Ist x kleiner als jedes Element in array , so behält left seinen Anfangswert und right erreicht schließlich left . Angenommen, der Anfangswert von left ist $\text{array}.\text{low}$. Dann gilt bei leerer Reihung: Anfangswert von $\text{right} = \text{array}.\text{high} = \text{array}.\text{low} - 1 < \text{array}.\text{low} = \text{Anfangswert von left}$, was der Schleifeninvariante $\text{left} \leq \text{right}$ widerspricht. Also *muss* der Anfangswert von left der nächstkleinere Wert $\text{array}.\text{low} - 1$ sein, der kein gültiger Reihungsindex ist. Ist also x kleiner als jedes Element in array , so unterschreitet right die Untergrenze des Reihungsindexbereichs. Zusammengefasst gilt: Ist die Reihung leer oder das gesuchte Element kleiner als das kleinste in der Reihung, so haben left und right nach der Schleife den ungültigen Indexwert $\text{array}.\text{low} - 1$. Deshalb muss nach der Schleife zuerst mit $\text{array}.\text{high} \leq \text{right}$ die Gültigkeit des Index right geprüft werden, bevor geprüft wird, ob das gesuchte Element in $\text{array}[\text{right}]$ steht.

Aufgabe 3

Lineare Suche
verifizieren

Finden Sie zur strukturierten Standardform des linearen Suchalgorithmus in Programm 28 geeignete Schleifeninvarianten und beweisen Sie die Korrektheit des Algorithmus! Hinweis: Suchen Sie in *Krimskrams – Folgen durchlaufen*!

Aufgabe 4

Einfache binäre Suche
verifizieren

Finden Sie zum einfachen binären Suchalgorithmus in Programm 34 geeignete Schleifeninvarianten und beweisen Sie die Korrektheit des Algorithmus! Hinweis: Suchen Sie in *Krimskrams – Binäre Suche*!

10**Alternativ abstrakt und konkret**

Leider passt die beschleunigte binäre Suche in Programm 36 nicht zur Abstraktion in Programm 22. Eine Abstraktion, die lineare, einfache und beschleunigte binäre Suche als Spezialisierungen erlaubt, lässt sich leicht aus Programm 36 extrahieren: Sie braucht eine abstrakte Schleifenabbruchbedingung (done) und eine abstrakte Bedingung für die Gefunden-oder-nicht-Entscheidung nach der Schleife (found). Von Programm 22 entfallen is_empty und item, init und reduce bleiben. Für einfache Suchverfahren muss done mit x parametrisiert sein, für die beschleunigte binäre Suche muss found mit x parametrisiert sein. Das Ergebnis zeigt Programm 38.

Programm 38

Oo Pseudocode
vertraglich spezifiziert,
partiell implementiert:
allgemeiner Suchraum

```

abstract class SEARCHSPACE_GENERAL [generic E]
public
  queries
    has (x : E) : BOOLEAN
    -- Template method using services of protected part.
  do
    init (x)
    until done (x) loop
      reduce (x)
    end
    return found (x)
  end
protected
  queries
    done (x : E) : BOOLEAN
    abstract
    found (x : E) : BOOLEAN
    pre
      done (x)
    abstract
  actions
    init (x : E)
    abstract
    reduce (x : E)
    pre
      not done (x)
    abstract
end

```

In welcher Beziehung stehen SEARCHSPACE von Programm 22 und SEARCHSPACE_GENERAL von Programm 38 zueinander? Jede der beiden Klassen lässt sich als Spezialisierung der anderen definieren. Aber SEARCHSPACE erlaubt weniger, SEARCHSPACE_GENERAL mehr Spezialisierungen. SEARCHSPACE_GENERAL als Spezialisierung von SEARCHSPACE führt zu fragwürdigen Entwurfssymptomen wie der Redefinition der Schablonenmethode has und der Stilllegung von is_empty und item.

Weitaus besser ist, SEARCHSPACE als Spezialisierung von SEARCHSPACE_GENERAL zu definieren. Damit die beiden Varianten von SEARCHSPACE unterscheidbar bleiben, heißt das SEARCHSPACE_GENERAL spezialisierende SEARCHSPACE in Programm 39

SEARCHSPACE_SPECIAL. Es führt `is_empty` und `item` als neue abstrakte Abfragen ein und implementiert damit die geerbten abstrakten Abfragen `done` und `found`. Damit weitere Erweiterungsklassen die Korrektheit dieser Implementationen nicht zerstören können, werden sie mit dem *Java* entlehnten **final** als endgültig deklariert.

Programm 39

Oo Pseudocode
vertraglich spezifiziert,
partiell implementiert:
spezieller Suchraum

```
class SEARCHSPACE_SPECIAL [generic E]
  extends SEARCHSPACE_GENERAL [generic E]
protected
  queries
    is_empty : BOOLEAN
      abstract
    item : E
      pre
        not is_empty
      abstract
    done (x : E) : BOOLEAN
      defined final
      do
        return is_empty or else item = x
      end
    found (x : E) : BOOLEAN
      pre
        -- done (x)                                -- from SEARCHSPACE_GENERAL
      defined final
      do
        return not is_empty
      end
end
```

Was ist damit erreicht? Die lineare Suche von `LINEAR_SEARCHARRAY` in Programm 23 und die einfache binäre Suche von `BINARY_SEARCHARRAY` in Programm 29 lassen sich nun auch als Konkretisierungen von `SEARCHSPACE_SPECIAL` in Programm 39 realisieren. Diese Varianten brauchen wegen der Vermittlerrolle von `done` und `found` etwas mehr Laufzeitaufwand als die Konkretisierungen von `SEARCHSPACE` in Programm 22, führen aber beim Verflachen und Expandieren zu identischen Programmen.

Programm 40 realisiert die beschleunigte binäre Suche von Programm 36 als Konkretisierung von `SEARCHSPACE_GENERAL` in Programm 38. Verflachen und Expandieren von `FAST_BINARY_SEARCHARRAY` in Programm 40 führt wieder zu Programm 36. (Der Leser möge diese Behauptungen selbst prüfen.) Programm 40 zeigt, dass auf Implementationen abstrakter Dienste verzettelte Algorithmenteile Korrektheitsbeweise erschweren. Sinnvollerweise führt man solche Umsetzungen mit Algorithmen durch, deren Korrektheit zuvor an einer kompakten prozeduralen Variante bewiesen wurde.

Die Abstraktion in Programm 22 haben wir durch eine Reihe von Modell- und Programmumformungen entwickelt, startend beim mathematischen Modell der Rekursionsformeln 3 und 4. Die Idee zur Abstraktion in Programm 38 entstand beim konkreten Algorithmus der beschleunigten binären Suche. Benötigt die Abstraktion in Programm 38 ein objektorientiertes Modell? Oder hätten wir sie auch in einem anderen Modell finden und formulieren können? Ja, wir hätten beispielsweise auch zuerst die Rekursionsformel 8 aufstellen und diese dann schrittweise in Programm 38 umformen können.

Rekursionsformel 8

Bedingter Ausdruck:
Enthält der Suchraum
das gesuchte
Element?

```
x ∈ S ⇔ if S = ∅ or S = {aS} then
          S ≠ ∅ and then aS = x
        else
          x ∈ S'
        end
```

Verbalisiert: Der Suchraum enthält das gesuchte Element genau dann, wenn er entweder höchstens einelementig ist und das gesuchte Element enthält, oder wenn der reduzierte Suchraum das gesuchte Element enthält.

Programm 40

Oo Pseudocode

implementierte

Abstraktion:

Beschleunigte binäre

Suchreihung

```

class FAST_BINARY_SEARCHARRAY [generic E extends COMPARABLE]
  extends SEARCHSPACE_GENERAL [generic E]
protected
  data
    array : ARRAY [E] range low .. high
    left, mid, right : INTEGER

  queries
    done (x : E) : BOOLEAN
      defined
      do
        return left >= right
      end

    found (x : E) : BOOLEAN
      pre
        -- done (x)                                -- from SEARCHSPACE_GENERAL
      defined
      do
        return left <= array.high and then array [left] = x
      end

    sorted (a : ARRAY [E]) : BOOLEAN
      -- body of program 29
      end

  invariants
    array.low <= array.high + 1
    sorted (array)
    array.low <= left
    left <= right
    right <= array.high + 1
    for all k = array.low .. left - 1 : array [k] < x
    for all k = right .. array.high : array [k] >= x

  actions
    init (x : E)
      defined
      do
        left := array.low
        right := array.high + 1
      end

    reduce (x : E)
      pre
        -- not done (x)                                -- from SEARCHSPACE_GENERAL
      defined
      do
        mid := (left + right) div 2
        if array [mid] < x then
          left := mid + 1
        else
          right := mid
        end
      end

end

```

Tabelle 4 bringt eine Übersicht zu den abstrakten Programmen 38 und 39 und den konkreten Programmen 23, 29 und 40.

Tabelle 4 Abstraktionen und Konkretisierungen

Klassen	Abstraktionen		Konkretisierungen		
	SEARCHSPACE_GENERAL Programm 38	SEARCHSPACE_SPECIAL Programm 39	LINEAR_SEARCHARRAY Programm 23	BINARY_SEARCHARRAY Programm 29	FAST_BINARY_SEARCHARRAY Programm 40
Daten			array index	array left, mid, right	array left, mid, right
Dienste: Abfragen	done abstract	defined is_empty or else item = x			defined left >= right
	found abstract	defined not is_empty			defined left <= array.high and then array [left] = x
		is_empty abstract	defined index > array.high	defined left > right	
		item abstract	defined array [index]	defined array [mid]	
Dienste: Aktionen	init abstract		defined index := array.low	defined left := array.low right := array.high mid := (left + right) div 2	defined left := array.low right := array.high + 1
	reduce abstract		defined index := index + 1	defined if item < x then left := mid + 1 else right := mid - 1 end mid := (left + right) div 2	defined mid := (left + right) div 2 if array [mid] < x then left := mid + 1 else right := mid end

Aufgabe 5

Lineare Suche in
Reihung mit
Stoppelement

Programm 41 zeigt eine beschleunigte Variante der **linearen Suche** in einer nichtleeren Reihung **mit Stoppelement** (*sentinel*) in prozeduraler Form. Finden Sie zu diesem Suchalgorithmus geeignete Schleifeninvarianten und beweisen Sie seine Korrektheit! Schätzen Sie seinen Laufzeitaufwand! Wie verhält er sich zum Laufzeitaufwand von Programm 28? Realisieren Sie diese Suchvariante als Konkretisierung FAST_LINEAR_SEARCHARRAY von SEARCHSPACE_GENERAL in Programm 38!

Programm 41

Prozeduraler
Pseudocode mit
generischer Funktion:
Lineare Suchreihung
mit Stoppelement

```
has [generic E] (array : ARRAY [E]; x : E) : BOOLEAN
  pre
    array.low <= array.high
  local
    index : INTEGER
    last_item : E
  do
    index := array.low
    last_item := array [array.high]
    array [array.high] := x
    until array [index] = x loop
      index := index + 1
    end
    array [array.high] := last_item
    return index < array.high or else last_item = x
  end
```

Aufgabe 6

Tabelle 4 ergänzen

Fügen Sie der Tabelle 4 drei Spalten hinzu, in die Sie die Elemente Ihrer Lösungen zu den Aufgaben 1, 2 und 5 eintragen!

Aufgabe 7

Beschleunigte binäre
Suche rekursiv
formulieren

Konstruieren Sie die beschleunigte binäre Suche in einer sortierten Reihung als rekursiven Algorithmus! Transformieren Sie die Lösung nach dem Schema zur Derekursivierung endrekursiver Funktionen (siehe [Skript Teil 1 Kapitel 3](#) S. 5) und prüfen Sie, ob die erhaltene iterative Variante äquivalent zu Programm 36 ist! Hinweis: Studieren Sie [Krimskrams – Binäre Suche](#)!

11**Gefunden, gelesen, geprüft**

Die meisten Programmierlehrbücher, die binäre Suche in Reihungen behandeln, bescheiden sich mit dem einfachen Algorithmus. Einige der seltenen Werke, die einen beschleunigten Binärsuchalgorithmus vorstellen, sind im Folgenden zitiert. Ihre Auswahl resultiert nicht aus einer umfassenden Literaturrecherche; ich habe auch nicht die aktuellsten Auflagen eruiert, in denen Fehler und Schwächen möglicherweise korrigiert sind. Im Text werden Fehler mit ☛ und Schwächen mit ☹ gelistet; bei den Programmen stehen ☛ für die Minimalzahl der Stellen, die zur Korrektur der Fehler geändert werden müssen, ☹ stehen für die Anzahl der Schwächen.

11.1**Wirth**

Niklaus Wirth stellt in seinem Lehrbuch *Algorithmen und Datenstrukturen* (Erstauflage 1975) lineare Suche ohne und mit Stoppelement und einfache und beschleunigte binäre Suche in Reihungen mit konstruktiven Erläuterungen, programmiersprachlichen Algorithmen, Aufwandsanalysen und Korrektheitsbeweisen verständlich, präzise und kompakt vor [Wir86] S. 55–58, [Wir04] S. 33–36. Wirth notiert die Algorithmen in seiner jeweils neuesten Sprache: *Pascal* [Wir83], *Modula-2* [Wir86], *Oberon* [Wir04].

Programm 42

Oberon: Beschleunigte
binäre Suche aus
[Wir04] S. 35

```
L := 0; R := N;
WHILE L < R DO
  m := (L+R) DIV 2;
  IF a[m] < x THEN L := m+1 ELSE R := m END
END
```

Für Korrektheitsbeweise stellt Wirth Schleifeninvarianten explizit und formal als prädikatenlogische Ausdrücke dar. Die softwaretechnisch fragwürdig kurzen Namen L, R, N, m erleichtern den Wechsel zwischen Programm und Programmbeweis. Ein Flüchtigkeitsfehler ($>$ statt $\geq$) in einer Schleifeninvariante der beschleunigten binäre Suche in der *Modula-2*-Version [Wir86] S. 58 ist in der *Oberon*-Version [Wir04] S. 35 korrigiert. Wirths Präsentation und Verifikation der Suchverfahren haben wesentlich zu Abschnitt 9 beigetragen. Da Wirths Buch viele weitere Algorithmen auf gleich hohem didaktischen und wissenschaftlichen Niveau beschreibt, sollte es trotz seines Alters Pflichtlektüre für Informatikstudenten sein.

11.2

Horowitz und Sahni

Ellis Horowitz und Sartaj Sahni konstruieren in ihrem Lehrbuch *Algorithmen. Entwurf und Analyse* (Erstauflage 1978) nach der „Teile-und-Herrsche“-Methode die einfache binäre Suche in Reihungen („Feld“), führen dazu Ablaufverfolgungen, einen Korrektheitsbeweis und gründliche Aufwandsanalysen durch, und stellen den iterativen Algorithmus `BINSUCHE1` zur beschleunigten binären Suche (Programm 43) und Ergebnisse vergleichender Laufzeitmessungen vor [HoS81] S. 125–133. Um sich „nicht mit den Besonderheiten und Eigentümlichkeiten einer [existierenden] Sprache [zu] belasten“, führen sie ihre an *Pascal* und *Algol 60* angelehnte Sprache *SPARKS* ein, deren Semantik operational durch Übersetzung in *Fortran* definiert ist [HoS81] S. 5, 754–764.

Programm 43
SPARKS:
Beschleunigte binäre
Suche aus [HoS81]
S. 131–132



```

procedure BINSUCHE1(A,n,x,j)
  //Gleiche Spezifikationen wie BINSUCHE außer: n> 0//
  integer unten, oben, Mitte, j, n;
  unten ← 1; oben ← n + 1 //"oben" ist immer um eins größer//
  while unten < oben - 1 do
    Mitte ← ⌊(unten + oben)/2⌋
    if x < A(Mitte) //nur ein Vergleich in der Schleife//
      then oben ←Mitte
    else unten ←Mitte //x ≥ A(Mitte)//
  endif
  repeat
    if x = A(unten) then j ←unten //x ist vorhanden//
    else j ←0 //x ist nicht vorhanden//
  until j ≠ 0
end BINSUCHE1

```

Die in Programm 43 referenzierte Spezifikation von `BINSUCHE` steht sechs Seiten vorher [HoS81] S. 125:

```

//entscheide zu gegebenem Feld A(1:n) mit Elementen in nichtab-//
//steigender Folge (n ≥ 0), ob x vorhanden ist. Wenn ja, setze //
//j so, daß gilt: A(j) = x, sonst setze j = 0.//

```

Horowitz und Sahni betonen: „Ein Korrektheitsbeweis ist viel wertvoller als tausend verschiedene Testläufe, denn er garantiert, daß das Programm korrekt arbeitet für alle möglichen Eingabewerte“ [HoS81] S. 5. Zu Programm 43 heißt es: „Der Korrektheitsbeweis für diesen Algorithmus bleibt einer Übungsaufgabe überlassen“ [HoS81] S. 131. Programm 43 ist korrekt, nur zwei Schwächen sind festzustellen:

- ⊗ Die Vorbedingung $n > 0$ verbietet das Suchen im leeren Suchraum. Nachteilig daran ist der Verlust an Allgemeinheit: Alle Kunden von `BINSUCHE1` müssen sicherstellen, dass *A* wenigstens ein Element enthält.
- ⊗ Programm 43 ist verglichen mit Programm 36 etwas schwerer verständlich, weil oben „immer um eins größer“ ist, die `while`-Bedingung aber mit `oben - 1` rechnet.

Die Algorithmen der Programme 36 und 43 unterscheiden sich auch darin, dass nach der Schleife in Programm 36 `left` den kleinsten Indexwert, in Programm 43 `unten` den größten Indexwert des Vorkommens des gesuchten Elements in der Reihung hat.

Aufgabe 8

Programm 43
verifizieren

Finden Sie für Programm 43 geeignete Schleifeninvarianten und beweisen Sie die Korrektheit des Programms!

11.3

Sedgewick

Robert Sedgewicks dickes Buch *Algorithmen* (Erstauflage 1983) begnügt sich mit einer iterativen Variante der einfachen binären Suche als *Pascal*-Prozedur `binarysearch` [Sed91] S. 233–241. Da es als Standardwerk gilt, ist es hier einbezogen. Obwohl die in *Algol 60* angelegten syntaktischen Schwächen von *Pascals* Steueranweisungen mit *Algol 68* Ende der 1960er und *Modula* Mitte der 1970er Jahre überwunden waren, benutzt Sedgewick noch in den 1990er Jahren *Pascal* statt die Nachfolgersprache *Modula-2*, weil „es weit verbreitet und sehr bekannt ist“ [Sed91] S. 8.

Programm 44

Pascal: Einfache
binäre Suche aus
[Sed91] S. 237



```
function binarysearch(v: integer): integer;
var x, l, r: integer;
begin
  l:=1; r:=N;
  repeat
    x:=(l+r) div 2;
    if v<a[x].key then r:=x-1 else l:=x+1
  until (v=a[x].key) or (l>r);
  if v=a[x].key
  then binarysearch:=x
  else binarysearch:=N+1
end;
```

Die Reihung `a` und die Variable `N` in `binarysearch` sind als globale Daten durch

```
var a: array[0..maxN] of node;
    N: integer;
```

vier Seiten vorher vereinbart. Die Voraussetzung „`a` sortiert“ steht in der Textzeile vor Programm 44. Zu `N` ist keine Einschränkung des Wertebereichs dokumentiert.

Sedgewick bietet viel Text, Abbildungen und eine Aufwandsschätzung, aber keinen Korrektheitsbeweis. Programm 44 enthält zwei Randwertfehler und vier Schwächen:

Inkonsistente Untergrenzen

- ☹ Das Suchintervall `l..r` wird mit `1..N` initialisiert, d.h. im Fall $N \geq 1$ ist das Reihungselement `a[0]` aus der Suche ausgeschlossen, das gesuchte Element `v` ist dort nicht findbar. Falls $N = 0$ oder $N = -1$, also das Suchintervall leer ist, wird `a[0].key` geprüft und vielleicht wird `v` beim Index `x = 0` gefunden. Ob diese drei Fälle so beabsichtigt sind, ist nach 14 Zeilen Erläuterung des Algorithmus unklar. Jedenfalls verwirren sie! Entwirren lassen sie sich durch einheitliche Untergrenzen von Reihung und Suchintervall mit einer dieser Zeilen:

```
a : array [0..maxN] of node  and  l := 0,
a : array [1..maxN] of node  and  l := 1.
```

Für die algorithmische Struktur ist übrigens belanglos, ob nach Reihungselementen oder Attributen von Reihungselementen gesucht wird. Daher stellt sich die Frage des didaktischen Nutzens des Attributs `key` von `node`.

Randwertfehler unten

- 💣 Im Fall $N < -1$ wird die Mitte $x < 0$, erhält also einen zu kleinen Indexwert für den Zugriff `a[x]`. Da *Pascal* die Gültigkeit von Reihungsindizes zur Laufzeit prüft, erkennt es den Programmierfehler zur Laufzeit. Ursache ist vermutlich, dass die Vorbedingung $N \geq -1$, oder, nach obiger Entwirrung der Untergrenzen, $N \geq 0$ bzw. $N \geq 1$, nicht dokumentiert ist.

Randwertfehler oben

- 💣 Falls $N > 2 * \text{maxN}$ wird $x > \text{maxN}$, erhält also einen zu großen Wert für `a[x]`. *Pascal* erkennt den Programmierfehler zur Laufzeit. Ursache ist vermutlich, dass die Vorbedingung $N \leq 2 * \text{maxN}$, oder besser stärker $N \leq \text{maxN}$, undokumentiert ist.

Fußgesteuerte
Bedingungsschleife

⊗ Die Verwendung der fußgesteuerten **repeat-until**-Schleife, die den Schleifenrumpf einmal gratis ausführt, ist unklar motiviert und erschwert das Verstehen und Verifizieren des Algorithmus. Eine weitere Verständnishürde ist, dass im Schleifenrumpf erst die Mitte x berechnet und danach die Grenze l oder r verschoben wird, und die Schleifenbedingung mit der alten Mitte und den neuen Grenzen rechnet.

Undokumentierter
Spezialwert

⊗ Der Codewert $N + 1$ für „nicht gefunden“ hängt erstens unnötigerweise von der Eingabegröße N ab, zweitens ist er nicht dokumentiert.

Schwache Namen

⊗ Die Namensgebung ist dürftig: Der Funktionsname `binsearch` bezeichnet den Algorithmus, sollte aber das Funktionsergebnis bezeichnen. Wieso die Mitte x heißt, ist rätselhaft. Das kleine l ist leicht mit 1 und r zu verwechseln, weshalb es als Name ungünstig wenn nicht ungeeignet ist.

Aufgabe 9

Programm 44
korrigieren, verifizieren

Beseitigen Sie die Fehler und Schwächen in Programm 44, dokumentieren Sie alle Bedingungen direkt beim Programm, finden Sie geeignete Schleifeninvarianten und beweisen Sie die Korrektheit Ihrer verbesserten Variante!

11.4

Ottmann und Widmayer

Auch Thomas Ottmann und Peter Widmayer beschränken sich in ihrem umfangreichen Lehrbuch *Algorithmen und Datenstrukturen* (Erstauflage 1990) auf die einfache binäre Suche, zeigen sie aber in verbaler, rekursiver und iterativer Form [OtW02] S. 165–168. Noch in der 4. Auflage von 2002 benutzen sie „eine um umgangssprachliche Mittel erweiterte Pascal-ähnliche Programmiersprache“ [OtW02] S. 2, deren Anweisungssyntax seit rund 30 Jahren veraltet ist.

Programm 45

Pascal: Einfache
binäre Suche aus
[OtW02] S. 167



```
function binsearch (k : integer) : integer;
{ liefert den Index eines Elements mit Schlüssel k im Bereich a[1], . . . ,a[N],
  falls es ein Element mit diesem Schlüssel gibt, und 0 sonst}
var m, l, r : integer;
begin
  l := 1; r := N;
  repeat
    m := (l + r) div 2;
    if k < a[m].key
    then r := m - 1
    else l := m + 1
  until (k = a[m].key) or (l > r);
  if k = a[m].key
  then binsearch := m
  else binsearch := 0
end
```

Da an Ottmanns und Widmayers iterativer Variante der binären Suche in Programm 45 die Ähnlichkeit mit Sedgewicks Programm 44 auffällt, ist es hier einbezogen. Die Erscheinungsjahre der Erstauflagen liefern Indizien für Vermutungen über Original und Plagiat. Von Programm 44 zu Programm 45 sind Namen geändert (`binsearch` zu `binsearch`, v zu k , x zu m), die Funktion durch einen Kommentar spezifiziert, der Nicht-gefunden-Wert $N + 1$ durch 0 ersetzt. Ottmann und Widmayer dokumentieren die Bedingungen an die globalen Daten klarer als Sedgewick. Zu den in `binsearch` vorkommenden Größen a und N steht zwei Seiten vorher die Vereinbarung

var a : array [0 .. $maxN$] **of** item;

und die Erklärung: „Die gegebenen N Elemente sollen an den Positionen $1, \dots, N$ stehen, $N \leq maxN$; jedes Element $a[i]$ hat eine Schlüsselkomponente $a[i].key$ “ [OtW02] S. 165. Die Untergrenze von N ist nicht angegeben, die Formulierung lässt $N \geq 0$ annehmen. Das Element $a[0]$, für die lineare Suche als Extraplatz für das Stoppelement vor-

gesehen, bleibt bei der binären Suche ungenutzt. Eine Seite vor Programm 45 steht die Vorbedingung von *binsearch*, dass *a* sortiert ist, mit der Erklärung

$$a[1].key \leq a[2].key \leq \dots \leq a[N].key.$$

Ottmann und Widmayer, die Korrektheit als „zweifellos“ „die wichtigste formale Eigenschaft eines Algorithmus“ werten [OtW02] S. 2, bieten eine Aufwandsschätzung, aber keinen Korrektheitsbeweis. Programm 45 halbiert die Mängel von Programm 44 auf einen Randwertfehler und zwei Schwächen:

Randwertfehler unten

- ☛ Falls $N = 0$, also bei leerem Suchintervall, wird die Mitte $m = 0$, sodass der Vergleich $k = a[m].key$ vielleicht das gesuchte Element k beim ungenutzten Index 0 findet.

Das Randwertfehlersymptom lässt sich zum Verschwinden bringen, indem in *binsearch* (a) die Spezifikation oder (b) die Implementation geändert wird. (a) erfolgt durch die Vorbedingung $N > 0$, die das Suchen im leeren Suchraum verbietet, und direkt bei *binsearch* zu dokumentieren ist. Nachteilig daran ist der Verlust an Allgemeinheit: Alle Kunden von *binsearch* müssen sicherstellen, dass *a* wenigstens ein Element enthält. (b) erfordert Änderungen in *binsearch* an drei Stellen.

Fußgesteuerte
Bedingungsschleife

- ⊗ Wie bei Programm 44 erschwert die Verwendung der **repeat-until**-Schleife das Verstehen und Verifizieren des Algorithmus, und die Reihenfolge der Zugriffe auf m , l und r bildet eine Verständnishürde.

Schwache Namen

- ⊗ Zwar sind k für *key* und m für die Mitte besser als v bzw. x , doch sonst ist die Namensgebung dürftig wie in Programm 44: Auch *binsearch* bezeichnet den Algorithmus statt das Funktionsergebnis. Das kursive kleine l ist zwar weniger leicht mit 1, aber noch leicht mit l zu verwechseln.

Aufgabe 10

Programm 45
korrigieren, verifizieren

Beseitigen Sie die Fehler und Schwächen in Programm 45, dokumentieren Sie alle Bedingungen direkt beim Programm, finden Sie geeignete Schleifeninvarianten und beweisen Sie die Korrektheit Ihrer verbesserten Variante!

Ottmanns und Widmayers rekursive Variante der einfachen binären Suche ist in Programm 46 zitiert. „Ein Aufruf [...] der Form *binsearch* (l, N, k) liefert [...] das gewünschte Ergebnis“ [OtW02] S. 166.

Programm 46

Pascalischer
Pseudocode:
Rekursive einfache
binäre Suche aus
[OtW02] S. 166



```

procedure binsearch ( $l, r, k : \text{integer}$ );
  {durchsuche  $a[l], \dots, a[r]$  nach einem Element mit Schlüssel  $k$ }
  var  $m : \text{integer}$ ;
  begin
     $m := (l + r) \text{ div } 2$ ;
    if  $l > r$ 
      then {Liste leer; Suche endet erfolglos}
    else
      begin
        if  $k < a[m].key$ 
          then binsearch( $l, m - 1, k$ )
        else if  $k > a[m].key$ 
          then binsearch( $m + 1, r, k$ )
        else { $a[m].key = k$ ; Suche endet erfolgreich}
      end
    end
  end

```

Der Randwertfehler von Programm 45 ist in Programm 46 verschwunden, sodass die Programme nicht äquivalent sind. Programm 46 schwächelt zweifach:

- ⊗ Die Ergebnisse des Algorithmus sind nur verbal durch Kommentare beschrieben. Informales kann Präzision mindern.
- ⊗ Warum ist die Einrücktiefe 6 bei logischer Schachtelungstiefe 3? Selbst die alte *Pascal*-Syntax erlaubt Einrückregeln, die logisch unnütze Einrückungen vermeiden.

Aufgabe 11
 Programm 46
 formalisieren,
 transformieren

Formalisieren Sie die verbalen Algorithmenteile in Programm 46, transformieren Sie *binsearch* nach dem Schema zur Derekursivierung endrekursiver Funktionen (siehe [Skript Teil 1 Kapitel 3](#) S. 5) und versuchen Sie, die erhaltene iterative Variante schrittweise in Programm 45 umzuformen! Bei welchem Umformungsschritt schleicht sich der Randwertfehler ein?

11.5

Appelrath und Ludewig

Hans-Jürgen Appelrath und Jochen Ludewig bringen in ihrem *Skriptum Informatik – eine konventionelle Einführung* nach der einfachen binären Suche in rekursiver und iterativer Form auch die beschleunigte binäre Suche in iterativer Form als *Modula-2-Prozedur Search* [ApL91] S. 287–291. Sie verwenden *Modula-2*, weil es als imperative Sprache keine „entrückte *Zweitwelt*“ einer „nicht-konventionellen Sprache“ aufbaut, sondern „die *Vorbereitung* des Objektbegriffs unterstützt“ und „praxistauglich und hinreichend verfügbar“ ist [ApL91] S. 3.

Programm 47
 Modula-2:
 Beschleunigte binäre
 Suche aus [ApL91]
 S. 290



```
PROCEDURE Search (      (* in *)   Wert: CARDINAL;
                       (* out *)  VAR gefunden: BOOLEAN;
                               VAR Pos: INTEGER);

VAR  Links, Rechts: INTEGER;
BEGIN (* Search *)
  Links := 0;  Rechts := n;
  WHILE Links < Rechts DO
    Pos := (Links + Rechts) DIV 2;
    IF F[Pos].Schluessel < Wert THEN
      Links := Pos + 1      (* rechten Bereich durchsuchen*)
    ELSE
      Rechts := Pos;        (* linken Bereich durchsuchen *)
    END (* IF *);
  END (* WHILE *);
  Pos := Rechts;  gefunden := (F[Pos] = Wert);
END Search;
```

Die in *Search* vorkommende Reihung *F* ist drei Seiten vorher vereinbart als

```
F : ARRAY [1..FLaenge] OF FeldTyp;
```

Dort wird auch die Variable *n* „als Index für das letzte tatsächlich belegte Element von *F*“ mit der Invariante „ $0 \leq n \leq \text{FLaenge}$ “ eingeführt [ApL91] S. 287. Die Vorbedingung von *Search*, dass *F* sortiert ist, steht verbal in der Erläuterung der binären Suche eine Seite vor dem Programm.

Appelrath und Ludewig geben Aufwandsschätzungen für die iterativen Varianten der binären Suche an, aber keinen Korrektheitsbeweis. Programm 47 enthält einen Flüchtigkeitsfehler und einen Konstruktionsfehler, der zwei Symptome bewirkt:

Statischer Typfehler

☛ Der Kompilierer erkennt in der zweitletzten Zeile von Programm 47 einen statischen Typfehler im Ausdruck *F[Pos] = Wert*, da *FeldTyp* und *CARDINAL* nicht kompatibel sind. Korrekt muss dort *F[Pos].Schluessel = Wert* stehen.

Randwertfehler unten

☛ Der Fall $n = 0$, die Suche im leeren Suchraum, versagt in Programm 47. Im Ablauf von *Search* mit $n = 0$ erhalten *Rechts* und *Pos* den Wert 0, sodass beim Zugriff *F[Pos]* die Indexuntergrenze 1 unterschritten wird. Da *Modula-2* die Gültigkeit von Reihungsindizes zur Laufzeit prüft, erkennt es den Programmierfehler zur Laufzeit. Wurde die Vorbedingung $0 \leq n \leq \text{FLaenge}$ beim Validieren vergessen, weil sie zu weit entfernt von der Funktion dokumentiert ist?

Das Randwertfehlersymptom verschwindet ähnlich wie jenes von Programm 45, indem in *Search* (a) die Spezifikation oder (b) die Implementation geändert wird. (a) erfolgt durch die Vorbedingung $n > 0$, die das Suchen im leeren Suchraum verbietet, und direkt bei *Search* zu dokumentieren ist. Nachteilig daran ist der Verlust an Allgemein-

heit: Alle Kunden von `Search` müssen sicherstellen, dass `F` wenigstens ein Element enthält. (b) erlaubt `n = 0` und erweitert den `gefunden` zugewiesenen Ausdruck zu `1 <= Pos AND F[Pos].Schluessel = Wert`, der bei `n = 0` den Wert `FALSE` ergibt.

Suchfehler unten

☛ Falls in Programm 47 `n > 0` und `Wert <= F[1].Schluessel` ist, also das gesuchte Element kleinergleich dem kleinsten Element im Suchraum ist, wird im Ablauf von `Search` im Schleifenrumpf stets der `ELSE`-Zweig ausgeführt, sodass `Rechts` stets kleiner und schließlich 1 wird, worauf `Pos` den Wert 0 erhält, infolgedessen beim Zugriff `F[Pos]` die Indexuntergrenze 1 unterschritten wird. *Modula-2* erkennt den Programmierfehler zur Laufzeit.

Ein Vergleich von Programm 47 mit Programm 36 zeigt, dass nicht Flüchtigkeit, sondern Fehlkonstruktion des Algorithmus das Suchfehlersymptom verursacht. Die Teile des Algorithmus passen nicht zusammen! Entweder ist der Schleifenrumpf korrekt, dann ist die Initialisierung falsch, oder umgekehrt. In beiden Fällen ist der `gefunden`-Ausdruck unvollständig. Um den Suchfehler zu beseitigen, sind drei Stellen zu ändern.

Aufgabe 12

Programm 47
korrigieren, verifizieren

Beseitigen Sie die Fehler in Programm 47, dokumentieren Sie alle Bedingungen direkt beim Programm, finden Sie geeignete Schleifeninvarianten und beweisen Sie die Korrektheit Ihrer verbesserten Variante!

11.6

Storer

In James A. Storer's Lehrbuch *An Introduction to Data Structures and Algorithms* dient die beschleunigte binäre Suche als Beispiel für Induktion und Rekursion [Sto02] S. 22, 89, 97. Storer orientiert sich beim Umsetzen seiner Idee

„Express algorithms with reasonably self-explanatory "pseudo-code" that mixes English with syntax that is equivalent to instructions available in most programming languages“ [Sto02] S. 4

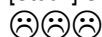
an der seit rund 30 Jahren schwächelnden pascalschen Anweisungssyntax und verzichtet auf Typisierung. Zum Problem

„**Problem:** Given part of an array $A[a] \dots A[b]$ that is already arranged in increasing order, and a value x , find an i , $a \leq i \leq b$, such that $x=A[i]$, or determine that no such i exists“ [Sto02] S. 22

gibt Storer eine iterative und eine rekursive Lösung an:

Programm 48

Pseudocode:
Beschleunigte binäre
Suche iterativ aus
[Sto02] S. 89



Iterative binary search algorithm:

```
while a < b do begin
  m := ⌊(a+b)/2⌋
  if x ≤ A[m] then b := m else a := m+1
end
if x=A[a] then {x is at position a} else {x is not in A}
```

Programm 49

Pseudocode:
Beschleunigte binäre
Suche rekursiv aus
[Sto02] S. 89



Recursive binary search algorithm:

```
function RBS(a,b)
  if a < b then begin
    m := ⌊(a+b)/2⌋
    if x ≤ A[m] then return RBS(a,m) else return RBS(m+1,b)
  end
  else if x=A[a] then return a
  else return -1
end
```

Vor Programm 49 steht die Erklärung: „Since we are assuming that $0 \leq a \leq b$, we use the convention that returning a value of -1 means that x is not in A “ [Sto02] S. 89. Storer untersucht Laufzeit- und Speicheraufwände und bringt zwei Korrektheitsbeweise: einen informalen zur iterativen Variante und einen formalen Induktionsbeweis zur rekursiven Variante. Die Programme 48 und 49 schwächeln nur schwach:

- ⊗ Die Vorbedingung $a \leq b$ verbietet das Suchen im leeren Suchraum. Nachteilig daran ist der Verlust an Allgemeinheit: Alle Kunden der Algorithmen müssen sicherstellen, dass A wenigstens ein Element enthält.
- ⊗ Programm 48 bleibt teilformal: Algorithmus, keine Funktion; Ergebnisse nur verbal durch Kommentare. Informales kann Präzision mindern.
- ⊗ Programm 48 verwendet a und b als initiale und variable Suchintervallgrenzen. Die Initialwerte gehen verloren, das kann verwirren.

Aufgabe 13

Programme 48 und 49 formalisieren, verifizieren, transformieren

Formalisieren Sie informale Teile in Programm 48, typisieren Sie die Programme 48 und 49, finden Sie geeignete Schleifeninvarianten zu Programm 48 und beweisen Sie seine Korrektheit, transformieren Sie Programm 49 nach dem Schema zur Derekursivierung endrekursiver Funktionen (siehe [Skript Teil 1 Kapitel 3](#) S. 5) und prüfen Sie, ob die erhaltene iterative Variante äquivalent zu Programm 48 ist! Falls das gesuchte Element x mehrfach in A vorkommt, welchen Index liefern die Lösungen?

11.7

Heun

Volker Heuns Lehrbuch *Grundlegende Algorithmen* enthält mehr Komplexitätsberechnungen als formale Algorithmen. Heun bemerkt, dass „man schon bei der Entwicklung eines Algorithmus auf dessen Korrektheit achten [muss], wobei es von einem korrekten Algorithmus zu einer korrekten Implementierung immer noch ein weiter Weg ist“ [Heu03] S. 1. Er will sich „mit dem Entwurf korrekter [...] Algorithmen beschäftigen“ [Heu03] S. 1 und „in der Regel“ vermeiden, „nur eine Beschreibung von Algorithmen anzugeben, ohne deren Korrektheit zu beweisen“ [Heu03] S. v.

Heuns „Algorithmen sind meist in einer der Programmiersprache C ähnlichen Notation verfasst, manchmal auch nur in Wörtern beschrieben, jedoch immer so präzise, dass eine Implementierung nicht weiter schwer fallen sollte“ [Heu03] S. vi. Warum *C*, das nicht nur wie *Pascal* unter der Erblast der *Algol 60*-Anweisungssyntax schwächelt, sondern an weiteren Mängeln kränkt? Was trägt die Präzision, mit der Kommas und Semikola an richtige Stellen platziert werden, zur logischen Korrektheit eines Algorithmus bei?

Programm 50

C-ähnlicher Pseudocode: Beschleunigte binäre Suche aus [Heu03] S. 34



```
BINARYSEARCH (int array[], int len, int elt)
{
    /* search in [l : r - 1] */
    int l = 0, r = len;
    for (int pos = (l + r)/2; l < r; pos = (l + r)/2)
        if (elt >= array[pos]) l = pos + 1;
        else r = pos;
    return l;
}
```

Heun gibt die beschleunigte binäre Suche in Programm 50 als Mittel zur Verbesserung des Sortierens durch Einfügen an, ohne seine Korrektheit zu beweisen. Dafür fehlt ohnehin die wesentliche Voraussetzung: eine explizite präzise Spezifikation.

Fehlende Spezifikation

- ⊗ Der größte Mangel an Programm 50 ist, dass BINARYSEARCH nicht explizit präzise spezifiziert ist. Die Spezifikation seines Ergebnisses lässt sich nur aus der Anwendung in INSERTIONSORT (nicht erläuterndem Text, sondern Pseudoanweisungen) erschließen: BINARYSEARCH(array, len, elt) liefert den Index, an der elt in array einzufügen ist, wenn array[0 : len - 1] ein sortiertes Anfangsstück ist. Die Vorbedingung

	„array[0 : len - 1] ist sortiert“ ist nicht explizit bei BINARYSEARCH dokumentiert, ebenso wenig die Vorbedingung, dass <i>len</i> die Reihungsgröße nicht überschreitet.																				
Unübliche Spezifikation	⊗ Da BINARYSEARCH nur implizit und anders als übliche Suchfunktionen spezifiziert ist, lässt es sich kaum in anderen Anwendungen benutzen – seine Wiederverwendbarkeit ist eingeschränkt. Missverständnisse über die Spezifikation sind zu erwarten. Dazu einige Beispiele: <table> <tr> <th>BINARYSEARCH(array, len, elt) Ergebnis</th><th>statt wie üblich</th></tr> <tr> <td>BINARYSEARCH({}, 0, elt)</td><td>0 <i>notFound</i></td></tr> <tr> <td>BINARYSEARCH({1}, 1, 0)</td><td>0 <i>notFound</i></td></tr> <tr> <td>BINARYSEARCH({1}, 1, 1)</td><td>1 0</td></tr> <tr> <td>BINARYSEARCH({1}, 1, 2)</td><td>1 <i>notFound</i></td></tr> <tr> <td>BINARYSEARCH({1, 3}, 2, 0)</td><td>0 <i>notFound</i></td></tr> <tr> <td>BINARYSEARCH({1, 3}, 2, 1)</td><td>1 0</td></tr> <tr> <td>BINARYSEARCH({1, 3}, 2, 2)</td><td>1 <i>notFound</i></td></tr> <tr> <td>BINARYSEARCH({1, 3}, 2, 3)</td><td>2 1</td></tr> <tr> <td>BINARYSEARCH({1, 3}, 2, 4)</td><td>2 <i>notFound</i></td></tr> </table>	BINARYSEARCH(array, len, elt) Ergebnis	statt wie üblich	BINARYSEARCH({}, 0, elt)	0 <i>notFound</i>	BINARYSEARCH({1}, 1, 0)	0 <i>notFound</i>	BINARYSEARCH({1}, 1, 1)	1 0	BINARYSEARCH({1}, 1, 2)	1 <i>notFound</i>	BINARYSEARCH({1, 3}, 2, 0)	0 <i>notFound</i>	BINARYSEARCH({1, 3}, 2, 1)	1 0	BINARYSEARCH({1, 3}, 2, 2)	1 <i>notFound</i>	BINARYSEARCH({1, 3}, 2, 3)	2 1	BINARYSEARCH({1, 3}, 2, 4)	2 <i>notFound</i>
BINARYSEARCH(array, len, elt) Ergebnis	statt wie üblich																				
BINARYSEARCH({}, 0, elt)	0 <i>notFound</i>																				
BINARYSEARCH({1}, 1, 0)	0 <i>notFound</i>																				
BINARYSEARCH({1}, 1, 1)	1 0																				
BINARYSEARCH({1}, 1, 2)	1 <i>notFound</i>																				
BINARYSEARCH({1, 3}, 2, 0)	0 <i>notFound</i>																				
BINARYSEARCH({1, 3}, 2, 1)	1 0																				
BINARYSEARCH({1, 3}, 2, 2)	1 <i>notFound</i>																				
BINARYSEARCH({1, 3}, 2, 3)	2 1																				
BINARYSEARCH({1, 3}, 2, 4)	2 <i>notFound</i>																				
Missbrauchte for-Schleife	⊗ Die for-Schleife, deren Semantik kaum intuitiv verständlich ist, dient in Programm 49 als reine kopfgesteuerte Bedingungschleife. Dafür bietet sich die while-Schleife an. Die for-Schleife sollte nur für Zählschleifen benutzt werden.																				
Unnötige Codeverdoppelung	⊗ Die Zuweisung $pos = (l + r)/2$ steht in der Schleifeninitialisierung und als Schleifenendeausdruck. Einmaliges Vorkommen am Schleifenrumpfanfang würde reichen.																				
Missbrauchter Schleifenendeausdruck	⊗ Der Schleifenendeausdruck sollte zählen. Die Variable <i>pos</i> ist kaum als Zähler zu betrachten, vielmehr ist $pos = (l + r)/2$ Teil einer Berechnung. Erlaubt man beliebige Ausdrücke als Schleifenendeausdrücke, so kann man die Schleife auch rumpfflos kurz schreiben: <pre>for (int pos; l < r; elt >= array[pos = (l + r)/2] ? l = pos + 1 : r = pos);</pre>																				
Partielle Typisierung	⊗ Während BINARYSEARCH beim Lesen zuerst als Prozedur erscheint, erkennt man bei return l, dass es eine Funktion ist. Aus dem Typ von <i>l</i> kann man auf den Ergebnistyp <i>int</i> schließen. Typisierung ist kein Muss in Pseudosprache, aber dient Typinferenz intuitiver Verständlichkeit?																				
Schwache Namen	⊗ Da BINARYSEARCH eine Funktion ist, passt sein Name nicht zum Funktionsergebnis. Zum kleinen <i>l</i> ist schon alles gesagt. Wozu ist <i>elt</i> verstümmelt?																				
Aufgabe 14 Programm 50 spezifizieren, korrigieren, verifizieren	Typisieren und spezifizieren Sie Programm 50 vollständig, beseitigen Sie seine Schwächen, finden Sie geeignete Schleifeninvarianten dazu und beweisen Sie die Korrektheit der Implementation gegenüber Ihrer Spezifikation!																				

11.8

Solymosi und Grude

Andreas Solymosi und Ulrich Grude widmen in ihrem dünnen *Grundkurs Algorithmen und Datenstrukturen* dem Suchen in einer Reihung vier Seiten, auf denen sie iterative Java-Funktionen für lineares und binäres Suchen mit Speicher- und Zeitbedarfen vorstellen [SoG00] S. 77–81. Solymosi und Grude wählen das unter der Erblast der C-Syntax schwächelnde Java als Notation, um „Studenten, die noch keine andere Programmiersprache beherrschen“, „Algorithmen und Datenstrukturen unterrichten zu können“ [SoG00] S. IX. Wurden Algorithmen nicht schon unterrichtet, als Studenten und Lehrer weder Java noch C, Pascal oder Algol beherrschten, sondern klares Denken in abstrakten Strukturen üben?

Da „die Informatikstudenten an Fachhochschulen [...] weniger an mathematischen Beweisen als an verständlich formulierten Algorithmen interessiert“ [SoG00] S. X. seien, erwähnt das Büchlein weder Verifikationsmethoden noch Begriffe wie Schleifeninvariante und Korrektheitsbeweis. Auf 15 Zeilen verbaler Beschreibung der einfachen binären Suche lassen Solymosi und Grude Programm 51 folgen:

Programm 51

Java: Beschleunigte
binäre Suche aus
[SoG00] S. 80–81



```
public int suchen(int suchSchluessel) {
    // ensures -1 <= return < sammlung.length
    int links = 0; int rechts = sammlung.length - 1 ;
    while (links < rechts) {
        int mitte = (links + rechts) / 2;
        if (suchSchluessel < sammlung[mitte].schluessel)
            rechts = mitte-1;
        else
            links = mitte+1;
    }
    links--; // zurücksetzen
    return suchSchluessel == sammlung[links-1].schluessel ? // gefunden
        links : -1; // nicht gefunden: -1
}
```

Doch Programm 51 entspricht nicht der einfachen binären Suche; die Schleifenbedingung deutet auf die beschleunigte Variante. Der Text vor Programm 51 erwähnt die Bedingung, „dass sammlung stets sortiert ist“. Daraus mag man schließen, dass die Invarianten `sammlung != null` sowie `sammlung[i] != null` für alle gültigen Indizes `i` erfüllt sind. Einfache Ablaufverfolgungen decken diese Fehler in Programm 51 auf:

- Suchfehler ☛ Bei leerer und einelementiger Reihung, `sammlung.length <= 1`, sollte `suchen(x)` -1 oder 0 liefern, wirft aber für jedes `x` eine `ArrayIndexOutOfBoundsException`.
- Suchfehler ☛ Im Fall `sammlung == {3, 5}` sollte `suchen(5)` den Index 1 liefern, wirft aber eine `ArrayIndexOutOfBoundsException`.
- Suchfehler ☛ Im Fall `sammlung == {3, 5, 5}` sollte `suchen(3)` den Index 0 liefern, wirft aber eine `ArrayIndexOutOfBoundsException`.

`suchen` findet weder oben noch unten enthaltene Schlüssel. Da Programm 51 vor Fehlern strotzt, haben es seine Autoren vermutlich keiner Ablaufverfolgung, keinem Korrektheitsbeweis und keinem Test unterzogen. Auch die 4. Auflage des Lehrbuchs von 2008 enthält noch alle Fehler bis auf einen. Zum Beseitigen der Fehlerursachen sind in Programm 51 sechs Stellen zu ändern, und zwar in der Initialisierung, im Schleifenrumpf und in der Restarbeit nach der Schleife. Keines der drei Teile ist ein korrekter Anfang, von dem aus die anderen Teile konstruiert werden könnten. Hier wurden nicht wie bei Programm 47 korrekte Teile falsch kombiniert und ein Randfall vergessen, sondern defekte Teile willkürlich montiert. Zudem sind als Schwächen anzumerken:

- Falsch dokumentierter Spezialwert ☹ Die Dokumentation des Codewerts -1 für „nicht gefunden“ ist wichtig, um die Funktion `suchen` benutzen zu können. Deshalb gehört sie in die Schnittstelle, nicht in die Implementation der Funktion.
- Schwacher Name ☹ Der Funktionsname `suchen` bezeichnet eine Aktion statt das Funktionsergebnis.

Aufgabe 15

Programm 51
korrigieren, verifizieren

Beseitigen Sie die Fehler und Schwächen in Programm 51, dokumentieren Sie alle Bedingungen direkt beim Programm, finden Sie geeignete Schleifeninvarianten und beweisen Sie die Korrektheit Ihrer verbesserten Variante!

11.9**Fazit**

Die obigen Abschnitte haben Darstellungen der binären Suche in den Programmen 42 bis 51 aus einer nicht repräsentativen Auswahl von Lehrbüchern untersucht. Welche Erkenntnisse lassen sich daraus gewinnen? Die Faktenbasis ist zu schmal, um Hypothesen zu formulieren. Die Ergebnisse regen aber zu Fragen und Programmierleitlinien an.

Fakten

Untersucht wurden 10 Varianten der binären Suche aus 8 Lehrbüchern, darunter 3 Varianten der einfachen binären Suche aus 2 Büchern und 7 Varianten der beschleunigten binären Suche aus 6 Büchern. Die einfache binäre Suche kommt einmal rekursiv, zweimal iterativ vor, die beschleunigte binäre Suche einmal rekursiv, sechsmal iterativ.

Qualität

Bei der Interpretation der Fakten interessiert, ob es Beziehungen zwischen der Qualität und anderen Merkmalen der Programme gibt. Die Untersuchung unterscheidet zwischen Fehlern (☹) und Schwächen (⊗). Schwächen bleiben im Folgenden unbeachtet, Qualität wird auf ein Gut-schlecht-Raster vergrößert:

gut := kein ☹-Fehler,
schlecht := mindestens ein ☹-Fehler.

Tabelle 5

Qualität und Form

Qualität	Form				gesamt
	einfach		beschleunigt		
	rekursiv	iterativ	rekursiv	iterativ	
gut	1	-	1	3	5
schlecht	-	2	-	3	5
gesamt	1	2	1	6	10

Form

Tabelle 5 fragt nach Beziehungen zwischen der Form der binären Suche und der Qualität ihrer Darstellung. Die kleinen Zahlen erlauben keine Interpretation. Dass die beiden rekursiven Varianten der binären Suche von guter Qualität und in Pseudosprache notiert sind, kann zufällig sein.

Tabelle 6

Qualität und Alter

Qualität	Erstauflage		gesamt
	bis 1999: alt	ab 2000: neu	
gut	3	2	5
schlecht	3	2	5
gesamt	6	4	10

Alter

Tabelle 6 fragt nach Beziehungen zwischen dem Alter eines Lehrbuchs und der Qualität der Darstellung der binären Suche, wobei ein grobes Raster für „alt“ und „neu“ genügt. Sie führt zur Frage: Variiert die Qualität von Darstellungen der binären Suche unabhängig vom Alter der Lehrbücher zwischen gut und schlecht?

Tabelle 7

Qualität und Notation

Qualität	Notation		gesamt
	Pseudosprache	Implementationssprache	
gut	4	1	5
schlecht	1	4	5
gesamt	5	5	10

Notation

Tabelle 7 fragt nach Beziehungen zwischen der Notation und der Qualität der Darstellung der binären Suche. Sie führt zu den Fragen: Trägt Pseudosprache eher positiv, Implementationssprache eher negativ zur Qualität von Darstellungen der binären Suche bei? Falls ja, warum? Um diese Fragen zu vertiefen, differenzieren wir die schlechte Qualität nach Art und Schwere der Fehler in Tabelle 8.

Tabelle 8
Fehlerart und Notation

Fehlerart	Notation		gesamt
	Pseudosprache	Implementations- sprache	
leichter Übersetzungszeitfehler: statischer Typfehler	-	1 (1 bbS)	1
undokumentierte Vorbedingung: Randwertfehler	-	4 (3 ebS, 1 bbS)	4
undokumentierte Spezifikation	1 (1 bbS)	-	1
schwerer Konstruktionsfehler	-	8 (8 bbS)	8
gesamt	1	13	14

Alle 8 schweren Konstruktionsfehler treten bei der iterativen beschleunigten binären Suche in Implementationssprache auf. Die beschleunigte binäre Suche ist schwerer als die einfache binäre Suche zu verstehen, zu konstruieren und zu verifizieren. Die Verwendung einer Implementationssprache schützt nicht vor Konstruktionsfehlern. Implementationssprachen sind keine Wundermittel gegen Denkfehler. Kein Programm, in welcher Sprache es auch geschrieben ist, kann besser als sein Entwurf sein. Zu verbreiteten Implementationssprachen existieren komfortable integrierte Entwicklungsumgebungen mit vielfältigen Testhilfen. Doch was hilft es, mit einem Debugger in Abläufen eines fehlkonstruierten, unverständenen Programms zu stochern?

Leitlinie 1
Konstruieren statt
basteln

Willst du zu einem Problem eine funktionsfähige, qualitätvolle Programmlösung erhalten, so analysiere früh das Problem, entwerfe und konstruiere nach bewährten Methoden, prüfe den Entwurf, die Konstruktion, führe Ablaufverfolgungen zu einfachen Eingabedaten und Korrektheitsbeweise mit Stift und Papier durch, implementiere und teste spät.

Programme sind nicht korrekt, wenn oder weil sie in Lehrbüchern schwarz auf weiß gedruckt stehen. Jedem Autor können Fehler unterlaufen. Programme in Lehrbüchern können Flüchtigkeitsfehler wie Übertragungsfehler beim Kopieren von Code aus Quellprogrammdateien in Buchdokumentdateien, Tippfehler, Syntaxfehler, aber auch logische Fehler wie Spezifikations- und Konstruktionsfehler enthalten.

Leitlinie 2
Kritikfähigkeit üben

Sei kritisch! Glaube Geschriebenes so wenig wie Gesprochenes! Verlass dich nicht darauf, dass veröffentlichte Programme ablaufverfolgt, verifiziert, übersetzt, getestet sind! Prüfe selbst, was du liest und hörst!

12

Implementiert

Nach 51 Programmen in verschiedenen Notationen sollte es, Heun paraphrasierend, „nicht weiter schwer fallen“, die linearen und binären Suchalgorithmen in Implementationssprachen zu übertragen und die Implementationen zu testen. Das Dokument *Krimskrams – Binäre Suche* enthält prozedurale generische endrekursive in C++ implementierte Varianten der binären Suche in Reihungen.

Aufgabe 16
Implementieren,
testen, erweitern

- (1) Übertragen Sie die Konzeptklassen SEARCHSPACE_GENERAL (Programm 38) und SEARCHSPACE_SPECIAL (Programm 39) und die Konkretisierungen LINEAR_SEARCHARRAY (Programm 23), FAST_LINEAR_SEARCHARRAY (Programm 41 in Aufgabe 5), BINARY_SEARCHARRAY (Programm 29) und FAST_BINARY_SEARCHARRAY (Programm 40) in die Implementationssprachen
 - (a) *Eiffel*,
 - (b) *Java*,

- (c) C#,
- (d) C++!

Übertragen Sie nicht nur Algorithmen und Daten, sondern auch vertragliche Spezifikationen und sonstige Zusicherungen!

- (2) Verifizieren Sie die implementierten Suchverfahren!
- (3) Entwerfen und implementieren Sie einen Testtreiber zum Testen der konkreten Klassen! Der Testtreiber benutzt nur die Schnittstelle der Konzeptklasse SEARCHSPACE_GENERAL über polymorphe Größen, an die Objekte der konkreten Klassen dynamisch gebunden werden. Der Testtreiber testet die Suchabfrage has. Die Erzeugung der Objekte ist in einer Fabrikfunktion gekapselt. Der Objekttyp ist beim Starten eines Tests wählbar.
- (4) Erweitern Sie die konkreten Klassen um Aktionen, die die Reihung mit Elementen füllen und sortieren! Der Testtreiber kann Objekte testen, deren Reihung mit Zufallselementen gefüllt ist. Da vier konkrete Klassen eine Reihung und die Aktionen dazu benötigen, sollten sie nicht repliziert, sondern in eine konkrete Klasse OWNER_OF_ARRAY herausfaktoriert werden. In *Eiffel* und *C++* bietet sich mehrfaches Erben zum Vereinigen der Suchraumabstraktion mit der Reihungseignerklasse an. In einem guten Entwurf wird OWNER_OF_ARRAY nur als geschütztes Mixin hinzugeerbt.

In *Java* und *C#* bildet man mehrfaches Erben mit etwas Schreibaufwand nach, indem man OWNER_OF_ARRAY in ein Interface OWNER_OF_ARRAY_ITFC und eine zugehörige Implementationsklasse OWNER_OF_ARRAY_IMPL zerlegt. Eine konkrete Klasse erweitert die Suchraumabstraktion und implementiert das Reihungseigner-Interface, indem es Aufrufe der Interface-Dienste an ein eigenes geschütztes Objekt der Reihungseignerklasse weiterleitet.

Da es potenziell viele Konkretisierungen zu Vereinigungen von Suchraumabstraktion und Reihungseigner gibt, lohnt sich dafür eigene Klassen, z.B.:

```
abstract class SEARCHARRAY_GENERAL [generic E]
extends
    SEARCHSPACE_GENERAL [generic E],
    OWNER_OF_ARRAY [generic E]
end
```

In einem guten Entwurf ist SEARCHARRAY_GENERAL eine *Schnittstellenklasse*, d.h. eine abstrakte Klasse mit einer vollständigen Schnittstelle, die alle zum Benutzen notwendigen Dienste bietet, sodass Konkretisierungen als *Implementationsklassen* auftreten: Sie müssen nur noch diese Dienste implementieren, aber keine weiteren Dienste hinzufügen.

- (5) Testen Sie alle konkreten Klassen intensiv mit verschiedenen Suchraumkapazitäten von 0 bis einige tausend!
- (6) Erweitern Sie den Testtreiber so, dass er Objekte mit verschiedenen aktuellen generischen Parametertypen testet! Wiederholen Sie Schritt (5)!
- (7) Entwerfen und implementieren Sie Konkretisierungen zu den Abstraktionen, die den Suchraum mit
 - (a) einer verketteten Liste (Aufgabe 1),
 - (b) einem geordneten Binärbaum (binären Suchbaum) (Aufgabe 2)
 implementieren! Erweitern Sie den Testtreiber entsprechend und wiederholen Sie Schritt (5)!
- (8) Beim Suchen eines Elements in einer Reihung ist oft nicht nur gefragt, ob sie das Element enthält, sondern genauer bei welchem Index. Ein mit zwei Schablonenmethoden partiell implementiertes Schnittstellenfragment steht in Programm 52. Lässt sich dieses Fragment in das schon bestehende Programm integrieren, ohne

eine weitere Klasse zu definieren? Falls ja, in welche Klasse? Welche technische, sprachabhängige Schwierigkeit entsteht beim Integrieren des Fragments in die existierende Klasse? Wie löst sich diese Schwierigkeit in den verschiedenen Implementationssprachen? Führen Sie die Integration des Fragments durch, erweitern Sie den Testtreiber so, dass er auch `some_index` testet, und wiederholen Sie Schritt (5)!

Programm 52

OoPseudocode partiell implementiert: Bei welchem Index enthält die Reihung das gesuchte Element?

```

public
  queries
    not_found : INTEGER = -- constant value which is never used as array index.
    some_index (x : E) : INTEGER
      -- Template method using services of protected part.
      do
        init (x)
        until done (x) loop
          reduce (x)
        end
        result := position (x)
      post
        -- result = some index such that array [index] = x, if any, else not_found.
      end

    has (x : E) : BOOLEAN
      -- Template method derived from some_index.
      do
        result := some_index (x) # not_found
      end

  protected
    data
      array : ARRAY [E] range low .. high

    queries
      done (x : E) : BOOLEAN
        abstract

      position (x : E) : INTEGER
        pre
          done (x)
        abstract
        post
          result # not_found implies array [result] = x
        end

    invariants
      not_found < array.low or array.high < not_found

    actions
      init (x : E)
        abstract

      reduce (x : E)
        pre
          not done (x)
        abstract

```

A

Literatur

- [ApL91] Hans-Jürgen Appelrath, Jochen Ludewig: *Skriptum Informatik – eine konventionelle Einführung*. Teubner, Stuttgart (1991) 448 S.
- [Fow99] Martin Fowler (with contributions by Kent Beck, John Brant, William Opdyke, Dan Roberts): *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Reading (1999) 431 S.

- [Fow00] Martin Fowler (mit Beiträgen von Kent Beck, John Brant, William Opdyke, Dan Roberts): *Refactoring. Wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, München (2000) 440 S.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*. Addison-Wesley, Reading (1994) 416 S.
- [GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn (2004) 504 S.
- [Heu03] Volker Heun: *Grundlegende Algorithmen. Einführung in den Entwurf und die Analyse effizienter Algorithmen*. Vieweg, GWV Fachverlage, Wiesbaden (2003) 2. verbess. u. erweit. Aufl., 370 S.
- [HoS81] Ellis Horowitz, Sartaj Sahni: *Algorithmen. Entwurf und Analyse*. Springer, Berlin, Heidelberg, New York (1981) 770 S.
- [Hug01] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion*. Vieweg, Braunschweig, Wiesbaden (2001) 2. Aufl., 446 S.
- [Mey09] Bertrand Meyer: *Touch of Class. Learning to Program Well with Objects and Contracts*. Springer, Berlin, Heidelberg, New York (2009) 876 S.
- [OtW02] Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukturen*. Spektrum, Heidelberg, Berlin (2002) 4. Aufl., 716 S.
- [Par83] David Lorge Parnas: *A Generalized Control Structure and Its Formal Definition*. Comm. ACM, Vol. 26, No. 8 (August 1983) S. 572–581, <http://dl.acm.org/citation.cfm?id=358168> (Zugriff 2012-10-03)
- [Sed91] Robert Sedgewick: *Algorithmen*. Addison-Wesley, Bonn (1991) 742 S.
- [SoG00] Andreas Solymosi, Ulrich Grude: *Grundkurs Algorithmen und Datenstrukturen. Eine Einführung in die praktische Informatik mit Java*. Vieweg, Braunschweig, Wiesbaden (2000) 194 S.
- [Sto02] James A. Storer: *An Introduction to Data Structures and Algorithms*. Birkhäuser, Boston, Springer, New York (2002) 599 S.
- [Wir83] Niklaus Wirth: *Algorithmen und Datenstrukturen. Pascal-Version*. Teubner, Stuttgart (1983) 3. überarbeit. Aufl., 320 S.
- [Wir86] Niklaus Wirth: *Algorithmen und Datenstrukturen mit Modula-2*. Teubner, Stuttgart (1986) 4. neubearbeit. u. erweit. Aufl., 299 S.
- [Wir04] Niklaus Wirth: *Algorithms and Data Structures. Oberon version*. (August 2004) 182 S., <http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf> (Zugriff 2012-10-03)