

# Einführung in die Programmierung

ETH Zürich

Datum: 6.2.2017

Nachname, Vorname: .....

Leginummer: .....

Ich bestätige mit meiner Unterschrift, dass ich die Prüfung unter regulären Bedingungen schreiben konnte und dass ich die untenstehenden Anweisungen gelesen und verstanden habe.

Unterschrift: .....

Anweisungen:

- Prüfungsdauer: 120 Minuten.
- Ausser einem Deutsch-Englisch Wörterbuch sind keine Hilfsmittel erlaubt.
- Es darf **nicht** mit Bleistift geschrieben werden.
- Bitte schreiben Sie Ihre Leginummer auf **alle** Blätter.
- Alle Lösungen können direkt auf die Aufgabenblätter geschrieben werden. Falls Sie mehr Platz für die Lösungen benötigen, bitten Sie die Aufsicht um zusätzliches offizielles Papier. Sie dürfen **nicht** Ihr eigenes Papier benutzen.
- Es darf nur eine Lösung pro Aufgabe abgegeben werden. Ungültige Lösungsversuche sind klar durchzustreichen.
- Bitte schreiben Sie leserlich! Es können nur lesbare Lösungen korrigiert werden.
- Teilen Sie sich die Zeit gut ein. Beachten Sie dazu die Anzahl der Punkte, die für die einzelnen Aufgaben vergeben werden.
- Bitte melden Sie sich **sofort**, falls Sie sich während der Prüfung gestört fühlen.
- Alle Programmteile sollen in der vom Kurs verwendeten Programmiersprache abgefasst werden: Eiffel. Alle Fragen betreffend der Programmiersprache beziehen sich, sofern nicht anders angegeben, auf Eiffel.

**Viel Erfolg!**

Frage	Anzahl möglicher Punkte	Punkte
1	10	
2	16	
3	16	
4	10	
5	18	
Gesamt	70	

## 1 Multiple Choice (10 Punkte)

Entscheiden Sie für jede Aussage über Eiffel und ihre Methodologie, ob sie wahr (W) oder falsch (F) ist, und kreuzen Sie das entsprechende Kästchen an. Jede korrekte Antwort wird mit 0.5 Punkten gewertet, inkorrekte oder fehlende Antworten geben keine Punkte.

Beispiel:

- |   | <b>W</b>                            | <b>F</b>                            |            |
|---|-------------------------------------|-------------------------------------|------------|
| a. $2 \times 4 = 8$                                 | <input type="checkbox"/>            | <input type="checkbox"/>            | 0 Punkte   |
| b. "Rösti" ist eine spezielle Art von Wurst.        | <input checked="" type="checkbox"/> | <input type="checkbox"/>            | 0 Punkte   |
| c. C ist eine objektorientierte Programmiersprache. | <input type="checkbox"/>            | <input checked="" type="checkbox"/> | 0.5 Punkte |

- |  | <b>W</b>                 | <b>F</b>                 |  |
|--|--------------------------|--------------------------|--|
| 1. Ein Feature ist eine Abfrage (query) falls es ein Objekt modifiziert.   | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 2. Ein Namenskonflikt ist akzeptabel, weil Eiffel Feature-Überladung unterstützt.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 3. Für ein beliebiges Objekt <code>o</code> gibt der Featureaufruf <code>o.is_equal(o)</code> immer <code>True</code> zurück.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 4. Binäre Suchbäume erlauben Zugriff in log-Zeit ( $O(\log n)$ ) im schlimmsten Fall.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 5. Ein Objekt kann aufgeschoben (deferred) oder effektiv sein.   | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 6. Eine Nachbedingung muss vor und nach der Ausführung eines jeden Features zutreffen, welches den Klienten der Klasse zur Verfügung steht.                              | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 7. Eine Erzeugungsprozedur muss sicherstellen, dass die neue Instanz nach der Ausführung der Erzeugungsprozedur die Invariante der Klasse erfüllt.                       | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 8. Eine Prozedur welche zu <code>NONE</code> exportiert wird, kann als Erzeugungsprozedur benützt werden.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 9. Polymorphismus ist die Fähigkeit von Objekten, ihren Typen während der Laufzeit zu ändern.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 10. Enthält eine Neudefinition einer Routine eine neue Nachbedingung, muss diese Bedingung zusätzlich zur geerbten Nachbedingung gelten.                                 | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 11. Wenn <code>C</code> eine aufgeschobene Klasse ist, kann in einem Programm keine Entität mit dem statischen Typ <code>C</code> existieren.                            | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 12. Eine Schleifeninvariante darf zwischen der Ausführung von zwei Befehlen im Schleifenkörper verletzt werden.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 13. Der Aufruf der Routine <code>r</code> kann nur dann in einer unendlichen Rekursion enden, wenn der Körper von <code>r</code> einen Aufruf zu <code>r</code> enthält. | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 14. Eine leere Vorbedingung ist äquivalent zur Vorbedingung <code>False</code> .   | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 15. Eine aufgeschobene Klasse kann von einer effektiven Klasse erben.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 16. Eine expandierte (expanded) Klasse <code>C</code> kann ein Attribut vom Typ <code>C</code> haben.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 17. Eine aufgeschobene Klasse kann Attribute haben.  | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 18. Void Referenzen können das Ziel von einem erfolgreichen Aufruf sein.   | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 19. Eine Klasse kann immer alle Features ihrer direkten Elternklassen (parent classes) aufrufen.   | <input type="checkbox"/> | <input type="checkbox"/> |  |
| 20. Verschiedene generische Ableitungen derselben generischen Klasse sind immer konform zueinander.  | <input type="checkbox"/> | <input type="checkbox"/> |  |

## 2 Vererbung und Polymorphismus (16 Punkte)

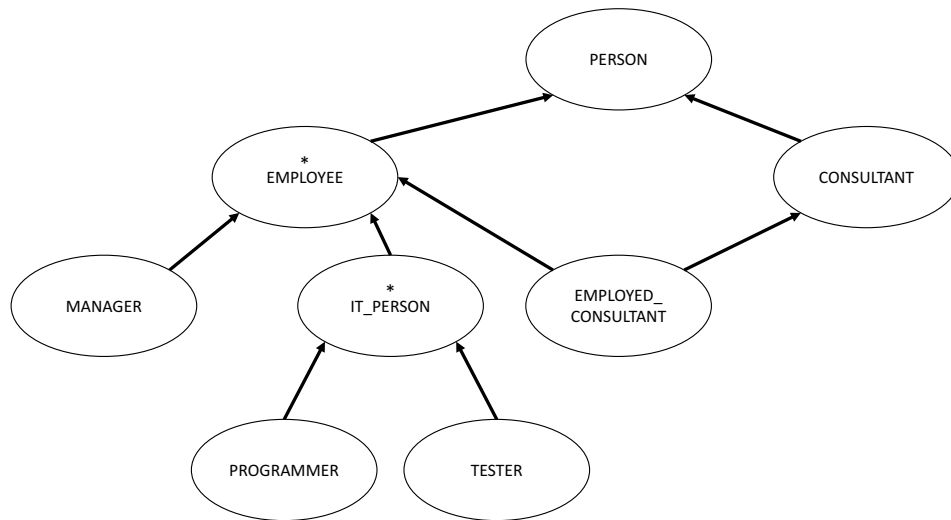


Figure 1: BON Diagram

Dieser Teil bezieht sich auf die folgenden Klassen:

```
class
  PERSON

create
  make

feature
  name: STRING

feature
  make (a_name: STRING)
    require
      a_name /= Void and then not a_name.is_empty
    do
      name := a_name
    ensure
      name = a_name
    end
end
```

```
deferred class
  EMPLOYEE

inherit
  PERSON

feature
```

```
    work
      deferred
    end
  end
end

class
  CONSULTANT

inherit
  PERSON
  rename
    make as make_person
  end

create
  make

feature
  consulting_topic: STRING

feature

  make (a_name: STRING; a_consulting_topic: STRING)
  do
    make_person (a_name)
    consulting_topic := a_consulting_topic
  end

  consult (a_person: PERSON)
  do
    print (name + " consults " + a_person.name + " on " + consulting_topic
          + ". ")
  end

  prepare
  do
    print (name + " prepares topic " + consulting_topic + ". ")
  end
end

class
  MANAGER

inherit
  EMPLOYEE
  rename
    work as manage,
    make as make_employee
  end

create
  make

feature
  subordinate: EMPLOYEE
```

```
feature
  make (a_name: STRING; a_subordinate: EMPLOYEE)
  do
    make_employee (a_name)
    subordinate := a_subordinate
    print ("Subordinate: " + subordinate.name + ". ")
  end

  manage
  do
    print (name + " issues a task for " + subordinate.name + ". ")
    subordinate.work
  end
end
```

```
deferred class
  IT_PERSON

inherit
  EMPLOYEE
  redefine
    make
  end

feature
  discuss_tech (a_colleague: IT_PERSON)
  do
    print (name + " discusses with " + a_colleague.name + ". ")
  end

  make (a_name : STRING)
  do
    precursor (a_name)
    print (a_name + " ")
  end
end
```

```
class
  EMPLOYED_CONSULTANT

inherit
  CONSULTANT
  rename
    prepare as study
  select
    make_person
  end

  EMPLOYEE
  rename
    make as make_employee,
    work as study
  end
```

```
create
  make
end
```

```
class
  PROGRAMMER
inherit
  IT_PERSON
  rename
    work as code
  redefine
    discuss_tech
  end
create
  make_c_programmer, make_eiffel_programmer

feature
  programming_language: STRING

feature
  make_c_programmer (a_name: STRING)
  do
    make (a_name)
    programming_language := "C"
    print ("programs in C. ")
  end

  make_eiffel_programmer (a_name: STRING)
  do
    make (a_name)
    programming_language := "Eiffel"
    print ("programs in Eiffel. ")
  end

  code
  do
    print (name + " writes some code. ")
  end

  discuss_tech (a_colleague: IT_PERSON)
  do
    precursor (a_colleague)
    print ("It is all about " + programming_language + ". ")
  end
end
```

```
class
  TESTER
inherit
  IT_PERSON
  rename
    work as test,
```

```
        make as make_it_guy
    end

create
    make

feature
    colleague: IT_PERSON

feature
    make (a_name: STRING; a_colleague: IT_PERSON)
    do
        make_it_guy(a_name)
        colleague := a_colleague
        print("has colleague " + colleague.name + ". ")
    end

    test
    local
        working_colleague : IT_PERSON
    do
        working_colleague := colleague
        if attached {TESTER} colleague as tester then
            working_colleague := tester.colleague
        end
        working_colleague.work
        print (name + " tests work from " + working_colleague.name + ". ")
    end
end
```

**Aufgaben** Geben Sie für die Code-Fragmente unten an, ob sie kompilieren oder nicht, indem Sie das entsprechende Kästchen ankreuzen. Falls das Code-Fragment nicht kompiliert, erklären Sie warum und markieren Sie die Zeile, die nicht kompiliert. Falls es kompiliert, geben Sie den Text an, welchen die Konsole ausgibt, sobald das Code-Fragment ausgeführt wurde.

Gegeben sind die folgenden Variablendeklarationen:

```
person: PERSON
employee: EMPLOYEE
it_person: IT_PERSON
manager: MANAGER
programmer: PROGRAMMER
tester1: TESTER
tester2: TESTER
consultant: CONSULTANT
employed_consultant: EMPLOYED_CONSULTANT
```

**Beispiel 1:**

```
create programmer.make_eiffel_programmer("Eric")
programmer.code
```

**Kompiliert der Code?**  Ja  Nein

**Ausgabe/Fehlerbeschreibung**

Eric programs in Eiffel. Eric writes some code.

### Beispiel 2:

```
create programmer.make_java_programmer("Jason")
```

Kompiliert der Code?  Ja  Nein

Ausgabe/Fehlerbeschreibung

Unknown identifier "make\_java\_programmer".

### Aufgabe 1

```
create it_person.make("Igor")
create programmer.make_eiffel_programmer("Eric")
it_person.discuss_tech(programmer)
```

Kompiliert der Code?  Ja  Nein

Ausgabe/Fehlerbeschreibung

.....  
.....  
.....

### Aufgabe 2

```
create programmer.make_eiffel_programmer("Eric")
employee := create {MANAGER}.make_manager("Max", programmer)
manager := employee
manager.manage
```

Kompiliert der Code?  Ja  Nein

Ausgabe/Fehlerbeschreibung

.....  
.....  
.....

### Aufgabe 3

```
create programmer.make_c_programmer("Chris")
it_person := create {PROGRAMMER}.make_eiffel_programmer("Eric")
it_person.discuss_tech(programmer)
```

Kompiliert der Code?  Ja  Nein

Ausgabe/Fehlerbeschreibung

.....  
.....

### Aufgabe 4

```
create {EMPLOYED_CONSULTANT} employee.make("Steve", "Data Model")
create person.make("Hector")
employee.work
```



Kompiliert der Code?  Ja  Nein  
Ausgabe/Fehlerbeschreibung

.....  
.....

### Aufgabe 5

```
it_person := create {PROGRAMMER}.make_c_programmer("Chris")  
it_person.code
```

Kompiliert der Code?  Ja  Nein  
Ausgabe/Fehlerbeschreibung

.....  
.....

### Aufgabe 6

```
create programmer.make_eiffel_programmer("Eric")  
create tester1.make("Ted", programmer)  
employee := create {MANAGER}.make("Max", tester1)  
employee.work
```

Kompiliert der Code?  Ja  Nein  
Ausgabe/Fehlerbeschreibung

.....  
.....  
.....  
.....

### Aufgabe 7

```
create programmer.make_c_programmer("Arnold")  
create tester1.make("Robert", programmer)  
create tester2.make("Delores", tester1)  
tester2.test
```

Kompiliert der Code?  Ja  Nein  
Ausgabe/Fehlerbeschreibung

.....  
.....  
.....  
.....

### 3 Softwarespezifikation durch Verträge (16 Punkte)

Ihr Start-up-Unternehmen entwickelt ein neues Webshop-Framework und Sie sind mit dem Backend der Software beauftragt. Sie haben sich entschieden, Design-by-Contract einzusetzen, um eine hohe Korrektheit Ihrer Software zu erreichen.

Hier sind einige Fakten:

- Eine frische Session startet immer mit einem leeren Warenkorb ohne spezifizierte Zahlungsmethode (payment method) und ohne Lieferadresse (shipping address).
- Nur positive Stückmengen können in den Warenkorb gelegt werden.
- Nur im Warenkorb existierende Gegenstände können entfernt werden.
- Sie können nur auschecken (check out) wenn die Zahlungsmethode ausgewählt wurde und die Lieferadresse gesetzt wurde.
- Bei der Anmeldung werden Zahlungsmethode und Lieferadresse auf die bevorzugten Werte eingestellt falls sie noch nicht gesetzt sind.
- Wenn es mindestens ein Produkt im Warenkorb gibt, welches mehr als 100 kostet, wird in der Session festgehalten, dass die nächste Bestellung einen Rabatt erhält. Ausserdem wird der Warenkorb geleert.

**Aufgabe** Die folgenden Klassen *ITEM* und *SESSION* sind die Herzstücke des Webshops. Bitte vervollständigen Sie die fehlenden Verträge (Vorbedingungen, Nachbedingungen und Klasseninvarianten), so dass alle Fakten der obigen informellen Spezifikation sowie andere nötige Eigenschaften abgedeckt sind.

- Die Anzahl der gepunkteten Linien repräsentiert nicht die Anzahl der fehlenden Vertragsklauseln.
- Sie müssen **True** überall dort einsetzen, wo kein Vertrag notwendig ist: Einen Vertrag leer zu lassen resultiert in 0 Punkten für diesen Abschnitt.
- Vergessen Sie nicht, alle Konsistenz-Bedingungen in den Invarianten zu formulieren. Sie sollten zum Beispiel die Wechselbeziehung zwischen dem Attribut *total* und dem Inhalt des Attributs *items* beschreiben.
- Das System verwendet keine Void-Safety. Entsprechend müssen **Void**-Referenzen explizit beachtet werden wo nötig.
- Neben anderen Features können Sie folgende Features der Klasse *HASH\_TABLE* annehmen:

```
class
  HASH_TABLE [G, K -> detachable HASHABLE]

feature

  for_all (test: FUNCTION [ANY, TUPLE [G,K], BOOLEAN]): BOOLEAN
    -- Is 'test' true for all key-value pairs?

  there_exists (test: FUNCTION [ANY, TUPLE [G,K], BOOLEAN]): BOOLEAN
    -- Is 'test' true for at least one key-value pair?
```

- Das folgende Feature der Klasse *CUSTOMER* könnte von Nutzen sein:

```
class CUSTOMER

feature
  preferred_payment: PAYMENT_METHOD
    -- The preferred payment method

  address: ADDRESS
    -- The address of the customer

  -- Other features omitted.
end
```

Listing 1: Class *ITEM*

```
class
  ITEM

inherit
  HASHABLE

create
  make

feature

  make (a_name: STRING; a_price: INTEGER)
    require

    .....

    .....

  do
    -- Implementation omitted.
  ensure

  .....

  .....

  .....

end

feature -- Access

  price: INTEGER
    -- Current price of the item.

  name: STRING
    -- Name of the item
```

```
feature -- Inherited

  hash_code: INTEGER
    -- Hash code value
  do
    -- Implementation omitted.
  end

  is_equal (a_other: like Current): BOOLEAN
  do
    -- Implementation omitted.
  end

invariant

.....

.....

.....

end
```

Listing 2: Class *SESSION*

```
class
  SESSION

create
  make

feature

  make
  require

  .....

  .....

  .....

  do
    -- Implementation omitted.
  ensure

  .....

  .....

  .....

  .....

  .....

  .....

end
```

**feature** -- Access

```
items: HASH_TABLE[INTEGER, ITEM]  
    -- Items in the basket, table from {ITEM} to {INTEGER}  
  
total: INTEGER  
    -- Current total price of all items in the basket.  
  
discount_next_order: BOOLEAN  
    -- If true, make a discount on the next order (should be set to true  
    -- upon checkout if there is at least one product costing more than 100)  
  
payment_method: PAYMENT_METHOD  
    -- The payment method selected by the customer.  
  
shipping_address: ADDRESS  
    -- Shipping addresses
```

**feature** -- Basic operations

```
add_item (a_quantity: INTEGER; a_item: ITEM)  
    -- Add 'a_quantity' 'a_item's to the basket.
```

**require**

```
.....  
.....  
.....
```

**do**

-- Implementation omitted.

**ensure**

```
.....  
.....  
.....  
.....  
.....
```

**end**

```
remove_item (a_quantity: INTEGER; a_item: ITEM)  
    -- Remove 'a_quantity' 'a_item's from the basket.
```

**require**

```
.....  
.....  
.....
```

**do**

-- Implementation omitted.

**ensure**

.....  
.....  
.....  
.....  
.....

**end**

*check\_out*

-- Check out the basket.

**require**

.....  
.....  
.....  
.....

**do**

-- Implementation omitted.

**ensure**

.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....

**end**

*set\_payment\_method* (*a\_payment\_method*: *PAYMENT\_METHOD*)

-- Set the payment method

**require**

.....  
.....  
.....

**do**

-- Implementation omitted.

```
ensure
.....
.....
.....
end

set_shipping_address (a_address: ADDRESS)
  -- Set the shipment address
  require
.....
.....
.....
do
  -- Implementation omitted.
  ensure
.....
.....
.....
end

log_in (a_customer: CUSTOMER)
  -- 'a_customer' has logged in
  require
.....
.....
.....
do
  -- Implementation omitted.
  ensure
.....
.....
.....
.....
.....
.....
.....
end
```

**feature** -- Extra space for features that you might need to create for writing the contracts

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

**invariant**

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

**end**



## 4 Rekursion: Catalan Nummern (10 Punkte)

Eine *Catalan Nummer* ist eine natürliche Zahl, welche Teil einer Sequenz ist, die in vielen interessanten Aufzählungsproblemen auftaucht.

Sie können eine Catalan Nummer mit der folgenden Formel berechnen:

$$C_0 = 1, \quad C_{n+1} = \sum_{i=0}^{i=n} C_i C_{n-i} \quad \text{for } n \geq 0$$

Durch das Anwenden der obigen Formel können Sie sehen, dass die ersten Catalan Nummern für  $n = 0, 1, 2, 3, 4, 5, 6, 7, \dots$  die folgenden sind: 1, 1, 2, 5, 14, 42, 132, 429, ...

**Aufgabe** Ihre Aufgabe ist es, eine rekursive Funktion *catalan\_number* zu implementieren, welche die Catalan Nummer  $C_n$  berechnet.

```
class CATALAN
```

```
    catalan_number (n: INTEGER): INTEGER
```

```
        -- Compute Catalan's number  $C_n$ .
```

```
    require
```

```
        n_non_negative: n >= 0
```

```
    local
```

```
        .....
```

```
        .....
```

```
    do
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
.....
```

end

## 5 Datenstrukturen: Begrenzte Prioritätswarteschlange (18 Punkte)

Ein Mobilfunkbetreiber will ein neues Near-Realtime-Monitoring-System einführen, um die 100 wertvollsten Kunden zu überwachen. Zu diesem Zweck speisen Sie eine leicht modifizierte *begrenzte Prioritätswarteschlange* mit einem Stream von Einnahmen, so dass die Warteschlange jederzeit die Einnahmen der aktuell Top 100 Kunden enthält.

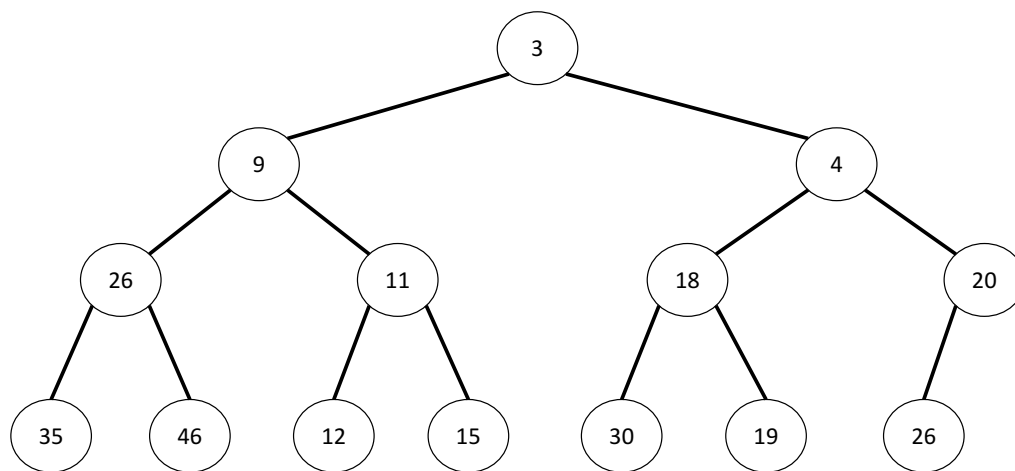
Eine typische Methode zur Implementierung einer Prioritätswarteschlange ist die Verwendung einer *Min-Heap*-Datenstruktur. Ein Min-Heap ist ein vollständiger binärer Baum, in dem der Wert eines Knotens kleiner ist als alle Werte in seinen Teilbäumen.

Heaps können auf sehr effiziente Weise (als implizite Datenstruktur) unter Verwendung eines Arrays dargestellt werden, ohne dass Zeiger zwischen Elementen erforderlich sind. Das erste Element enthält die Wurzel. Die nächsten beiden Elemente des Arrays enthalten ihre Kinder. Die nächsten vier enthalten die vier Kinder der beiden Kindknoten usw. Somit ist das linke Kind eines Knotens an der Position  $n$  an der Position  $2n$  und das rechte Kind ist immer rechts neben dem linken Kind. Dies ermöglicht es, durch einfache Indexberechnungen im Baum auf- und abwärts zu navigieren. Abbildung 2 zeigt ein Beispiel eines Binär-Heap und seine entsprechende Array-Darstellung.

Nachdem ein Element eingefügt oder ersetzt wurde, muss die Heap-Eigenschaft durch die internen Operationen *sift\_down* und *sift\_up* wiederhergestellt werden.

Bei der folgenden Implementierung wird das Feature *process* für alle Einnahmen aufgerufen. Dieses Feature fügt Einnahmen in der Warteschlange ein oder ersetzt diese, falls:

- die Warteschlange nicht voll ist, ODER
- die Warteschlange voll ist aber die aktuell verarbeiteten Einnahmen grösser sind als alle Einnahmen in der Warteschlange (wodurch die kleinsten Einnahmen aus der Warteschlange fallen)



3	9	4	26	11	18	20	35	46	12	15	30	19	26
---	---	---	----	----	----	----	----	----	----	----	----	----	----

Figure 2: Beispiel einer Prioritätswarteschlange, implementiert als binärer Heap, sowie dessen Array-Repräsentation.

**Aufgabe** Ihre Aufgabe besteht darin, die Implementierung der Funktionen *parent\_index*, *left\_child\_index*, *right\_child\_index* und eine effiziente, nicht-rekursiv Version von *sift\_down* zu vervollständigen.

Listing 3: Class *BOUNDED\_PRIORITY\_QUEUE*

```

class
  BOUNDED_PRIORITY_QUEUE [G -> COMPARABLE]

create
  make

feature {NONE}
  heap: ARRAY[G]
  max_size: INTEGER
  count: INTEGER

feature {NONE} -- Initialization

  make (n: INTEGER)
    -- Allocate heap space.
    do
      create heap.make(1, n)
      count := 0
      max_size := n
    end

feature -- Processing of new elements

  process (v: like item)
    -- Conditionally insert/replaces the item v in the queue.
    do
      if count < max_size then
        count := count + 1
        heap.force (v, count)
        sift_up
      elseif v > heap.item (1) then
        heap.put (v, 1)
        sift_down
      end
    end

feature {NONE} -- Private access parent/children indices

  parent_index (i: INTEGER): INTEGER
    -- Get index of parent of entry at position i
    do
      .....
      .....
    end

  left_child_index (i: INTEGER): INTEGER
    -- Get index of left child of entry at position i
  
```

```

do
    .....

end

right_child_index (i: INTEGER): INTEGER
    -- Get index of right child of entry at position i
do
    .....

end

feature {NONE} -- Heap internal operations

sift_down
    -- Sift top element down until the heap property holds again
    -- Efficient, non-recursive implementation required!
local
    .....

do
from
    .....

until
    .....

loop
    .....

    .....

    .....

    .....

    .....

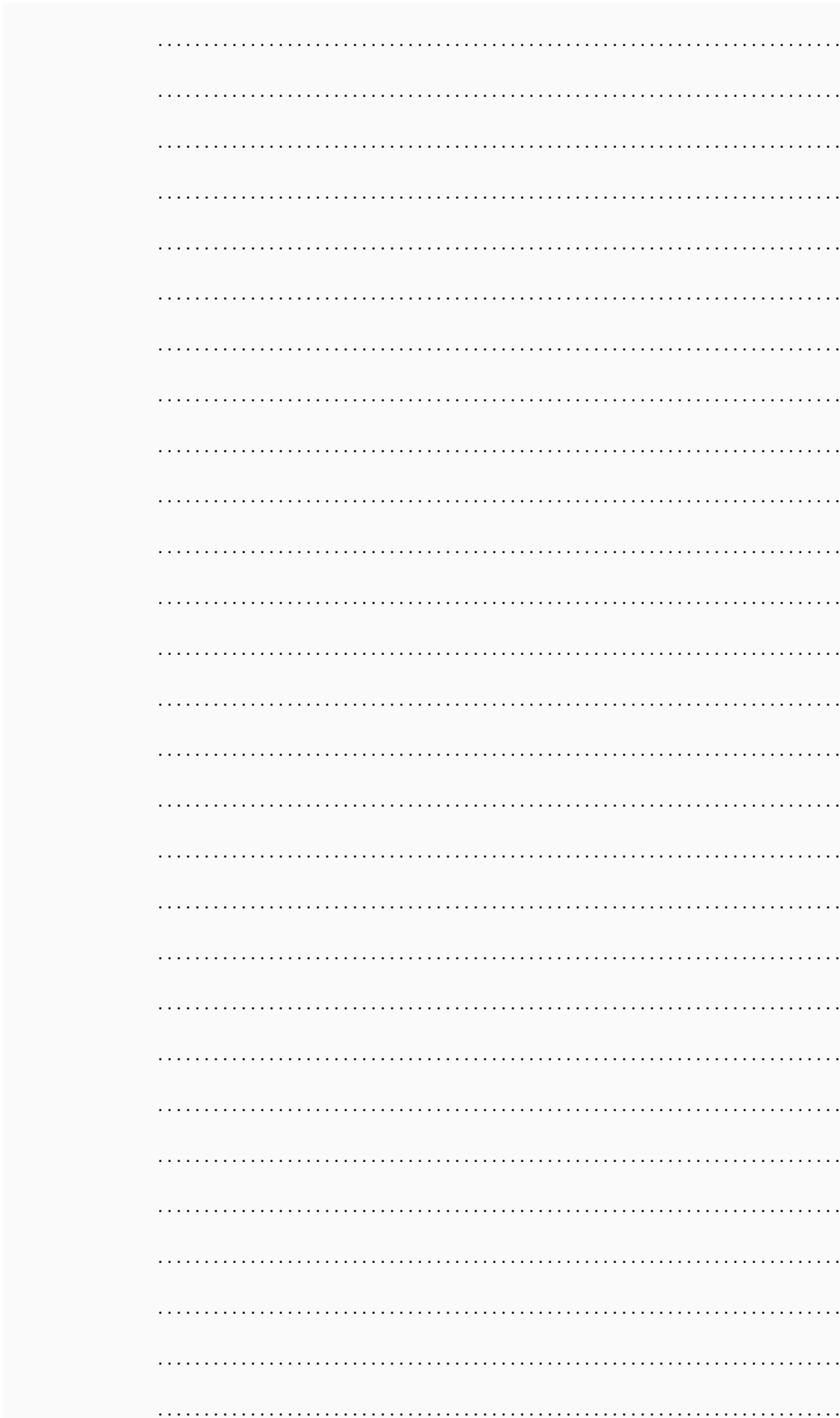
    .....

    .....

    .....

    .....

    .....
    
```



```
.....  
.....  
.....  
.....  
.....  
.....  
end  
  
.....  
.....  
.....  
.....  
.....  
.....  
end  
  
sift_up  
-- Sift last element up until the heap property holds again  
local  
  i, j: INTEGER  
  up, down: like item  
  stop: BOOLEAN  
do  
  from  
    i := count  
    down := heap.item (i)  
  until  
    stop or i = 1  
  loop  
    j := parent_index (i)  
    up := heap.item (j)  
    if up > down then  
      heap.put (up, i)  
      i := j  
    else  
      stop := true  
    end  
  end  
  heap.put (down, i)  
end  
  
end
```