Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr. Wolfgang Hommel
Michael Grabatin, Michael Steinke

WT 2020
Übungsblatt 9

Einführung in die Informatik II

13. und 16.03.2020

1 Highest In – First Out

In der Vorlesung wurden Formalisierungen für verschiedene Datenstrukturen gezeigt. Diese Aufgabe beschäftigt sich mit der Formalisierung für eine neue Datenstruktur: Highest In – First Out (HIFO).

Im Gegensatz zu einer Warteschlange (First In – First Out), die beim Abrufen eines Wertes immer den ältesten Wert in der Datenstruktur zurückgibt, soll die HIFO immer den größten verbleibenden Datensatz aus der Datenstruktur wiedergeben.

Die Operationen, die auf dieser Datenstruktur existieren, sind:

- `create`: Erzeugt eine neue leere Datenstruktur
- `add`: Fügt ein neues Element hinzu, sodass das größte Element jeweils am Anfang der Liste steht
- `get`: Gibt den Wert des größten Elementes in der Datenstruktur zurück
- `remove`: Entfernt das größte Element aus der Datenstruktur
- `empty`: Gibt an ob die Datenstruktur leer ist

a) Erstellen Sie die formale Signatur der oben beschriebenen Operationen. Verwenden Sie dazu E als Menge der Elemente, H als Menge der HIFO-Datenstrukturen und B als Menge der Wahrheitswerte.

> Die formale Signatur der Operationen sieht wie folgt aus. Die Operationen `get` und `remove`, werden auf leeren Listen nicht definiert.
>
> $$\begin{array}{ll|r} create & | & \rightarrow H \\ add & | & E \times H \rightarrow H \\ get & | & H - \{create\} \rightarrow E \\ remove & | & H - \{create\} \rightarrow H \\ empty & | & H \rightarrow B \end{array}$$

b) Erstellen Sie aus der Syntax der Operationen eine vollständige algebraische Definition. Woran erkennen Sie, dass Sie ausreichend aber nicht zu viele Axiome erstellt haben?

> Um einen vollständigen Satz an Axiomen zu erzeugen, braucht man für jeden Konstruktor Z und jeden Nicht-Konstruktor O ein Axiom der Art $O(Z(...)) = ...$ Konstruktoren sind $Z \in \{create, add\}$, Nicht-Konstruktoren sind $O \in \{get, remove, empty\}$. Eigentlich würden daher 6 Axiome benötigt, die Axiome $get(create)$ und $remove(create)$ sind allerdings nicht Sinnvoll, weshalb sie schon in der obigen Definition der Syntax ausgeschlossen wurden.
>
> $$\begin{array}{lll} (H1) & empty(create) & = & true \\ (H2) & get(add(e,h)) & = & if(h == create)\ e\ else\ \{\ if(e > get(h))\ e\ else\ get(h)\ \} \\ (H3) & remove(add(e,h)) & = & if(h == create)\ h\ else\ \{\ if(e > get(h))\ h\ else\ add(e, remove(h))\ \} \\ (H4) & empty(add(e,h)) & = & false \end{array}$$

c) Implementieren Sie die bislang nur formal spezifizierte Datenstruktur als abstrakten Datentyp. Die HIFO-Datenstruktur soll in diesem Fall Integer-Werte speichern.

Eine mögliche Implementierung wäre:

```
1  abstract class Hifo
2  case class Empty () extends Hifo
3  case class Elem (e: Int, h: Hifo) extends Hifo
4
5  def create(): Hifo = Empty()
6
7  def add(e: Int, h: Hifo): Hifo = h match {
8    case Empty()       => Elem(e, Empty())
9    case Elem(e2, h2) => if(e > e2) Elem(e, h)
10                        else Elem(e2, add(e, h2))
11 }
12
13 def get(h: Hifo): Int = h match {
14   case Empty()     => throw new Exception("Cannot get from Empty")
15   case Elem(e, h) => e
16 }
17
18 def remove(h: Hifo): Hifo = h match {
19   case Empty()     => throw new Exception("Cannot remove Empty")
20   case Elem(e, h) => h
21 }
22
23 def empty(h: Hifo): Boolean = h match {
24   case Empty()     => true
25   case Elem(_, _) => false
26 }
27
28 //// Test der Axiome und der einzelnen Fälle
29 // H1
30 empty(create()) == true                    //> res: Boolean = true
31
32 // H2
33 get(add(2, create())) == 2                 //> res: Boolean = true
34 get(add(2, add(5, create()))) == 5         //> res: Boolean = true
35 get(add(5, add(2, create()))) == 5         //> res: Boolean = true
36
37 // H3
38 remove(add(2, create())) == create()       //> res: Boolean = true
39 remove(add(2, add(5, create()))) == add(2, create())
40                                            //> res: Boolean = true
41 remove(add(5, add(2, create()))) == add(2, create())
42                                            //> res: Boolean = true
43
44 // H4
45 empty(add(4, create())) == false           //> res: Boolean = true
```

d) Erstellen Sie nun noch die Spezifikation der HIFO-Datenstruktur mit Zusicherungen. Geben Sie dazu zunächst die notwendigen Invarianten an.

Die notwendigen Invarianten sind ähnlich zu denen des Stack aus der Vorlesung.

$(I1)$ $\forall (e1, n1), (e2, n2) \in hifo:$ $(n1 = n2) \rightarrow (e1 = e2)$
$(I2)$ $\forall (e, n) \in hifo:$ $0 \leq n < |hifo|$
$(I3)$ $\forall (e1, n1), (e2, n2) \in hifo:$ $(n1 < n2) \rightarrow (e1 \geq e2)$

e) Spezifizieren Sie für die einzelnen Operationen, die auf der HIFO-Datenstruktur definiert sind, jeweils die Vor- (pre) und Nachbedingung (post).

$create$	$pre:$	
	$post:$	
$add(e, h)$	$pre:$	
	$post:$	
$get(h)$	$pre:$	
	$post:$	
$remove(h)$	$pre:$	
	$post:$	
$empty(h)$	$pre:$	
	$post:$	

Mögliche Spezifikation der Zusicherungen. h' beschreibt jeweils den Zustand der HIFO-Datenstruktur nach der Ausführung der Operation.

$create$	$pre:$	$true$
	$post:$	$h' = \emptyset$
$add(e, h)$	$pre:$	$true$
	$post:$	$h' = \{(e, 0)\} \cup \{(x, m+1) \mid (x, m) \in h\}$
$get(h)$	$pre:$	$\lvert h \rvert > 0$
	$post:$	$h' = h \wedge \exists (e, n) \in h : n = 0 \wedge get(h) = e$
$remove(h)$	$pre:$	$\lvert h \rvert > 0$
	$post:$	$\forall (e, n) : ((e, n-1) \in h' \leftrightarrow ((e, n) \in h \wedge n \geq 1))$
$empty(h)$	$pre:$	$true$
	$post:$	$h' = h \wedge empty(h) = (\lvert h \rvert = 0)$

2 Hashes

Diese Aufgabe widmet sich dem Thema Streuspeichertabellen und Hash-Funktionen.

In Java und somit auch in Scala ist für jeden Typ (auch für selbst definierte) automatisch eine Hash-Funktion (durch die Methode `hashCode`) definiert. Für eine Zeichenkette `s` wird der *Hash-Code* `h` (vom Typ `Int`) wie folgt berechnet:

```
1 var h = 0
2 for (c <- s) {
3   h = 31 * h + c
4 }
```

a) Gegeben sind folgende Namen:

Anna, *Bob*, *Dennis*, *Julia*, *Karl*, *Lena*, *Maike*, *Markus* und *Sarah*.

Bestimmen Sie für alle Namen gemäß obiger Berechnung die *Hash-Codes*. In Scala berechnet, weist der zugehörige *Hash-Code* für einen Namen eine Besonderheit auf. Erklären Sie diese.

> Die *Hash-Codes* lassen sich folgender Tabelle entnehmen:
>
> | **Name:** | ***Hash-Code:*** |
> |---|---|
> | Anna | 2045632 |
> | Bob | 66965 |
> | Dennis | 2043443979 |
> | Julia | 71933241 |
> | Karl | 2331184 |
> | Lena | 2364684 |
> | Maike | 74105167 |
> | Markus | -1997438389 |
> | Sarah | 79654635 |
>
> Der *Hash-Code* `hc1` für *Markus* ist negativ. Dies lässt sich wie folgt erklären:
>
> Der *Hash-Code* `hc2` von *Marku* beträgt 74113832, ist also noch positiv. `hc1` geht gemäß Algorithmus aus `hc2` durch folgende Berechnung hervor:
>
> ```
> 1 hc1 = 31 * hc2 + 's'
> ```
>
> Die Multiplikation bewirkt einen Integerüberlauf da $31 * 74113832 = 2297528792$, der *Hash-Code* aber vom Typ `Int` ist und daher maximal $2^{31} - 1 = 2147483647$ sein kann.

b) Die Namen sollen nun in eine „*offene*“ Tabelle der Länge $m = 30$ eingetragen werden. Bestimmen Sie für alle Namen auf Grundlage des berechneten *Hash-Codes* den jeweiligen „*primären Speicherort*“ in der Tabelle. Wählen Sie eine geeignete Behandlung des Sonderfalls aus Teilaufgabe *a*. Ist „*perfektes Hashing*“ möglich oder treten Kollisionen auf? Wenn nötig, nutzen Sie eine geeignete Kollisionsbehandlung.

> Für die Bestimmung des „*primären Speicherortes*“ s gilt bei gegebenem *Hash-Code* h und gegebener Länge m grundsätzlich folgende Berechnung:
>
> $s = h \ mod \ m$.
>
> Negativen *Hash-Codes* lässt sich wie folgt begegnen:
>
> $s1 = |h| \ mod \ m$ (Absolutbetrag)
>
> $s2 = h \ mod \ m + m$.
>
> Folgende Tabelle liefert die „*primären Speicherorte*“ der Namen für beide Varianten:

Name:	***Hash-Code:***	***s1***	***s2***
Anna	2045632	22	22
Bob	66965	5	5
Dennis	2043443979	9	9
Julia	71933241	21	21
Karl	2331184	4	4
Lena	2364684	24	24
Maike	74105167	7	7
Markus	-1997438389	19	11
Sarah	79654635	15	15

Offensichtlich liegen keine Kollisionen vor.

c) Die Namen aus Teilaufgabe *a* sollen in eine Tabelle der Größe $m \geq 35$ eingetragen werden. Für welches m ist erstmalig kein „*perfektes Hashing*“ möglich? Bestimmen Sie das $m' \geq 35$, bei dem mehr als zwei Kollisionen auftreten. Stellen Sie den Zustand einer „*geschlossenen*“ Tabelle der Länge m' dar, in welcher alle Namen eingetragen sind.

Für $m_1 = 35$ tritt bereits eine Kollision auf (der Speicherort 9 wird doppelt belegt). Bei $m_2 = 37$ treten 3 Kollisionen auf:

Name:	***Hash-Code:***	***s1***	***s2***
Anna	2045632	13	13
Bob	66965	32	32
Dennis	2043443979	24	24
Julia	71933241	24	24
Karl	2331184	36	36
Lena	2364684	14	14
Maike	74105167	13	13
Markus	-1997438389	12	25
Sarah	79654635	36	36

Bei einer geschlossenen Tabelle wird eine externe Kollisionsbehandlung verwendet, die kollidierenden Elemente also in einer linearen Liste gehalten. Folgende Situation ergibt sich:

Position:	**Elemente nach *s1***	**Elemente nach *s2***
12	*Markus*	-
13	*Anna*, *Maike*	*Anna*, *Maike*
14	*Lena*	*Lena*
24	*Dennis*, *Julia*	*Dennis*, *Julia*
25	-	*Markus*
32	*Bob*	*Bob*
36	*Karl*, *Sarah*	*Karl*, *Sarah*

3 Maps

In dieser Aufgabe geht es darum, die Map-Datenstruktur beispielhaft in Scala zu implementieren. Eine Map dient der Speicherung von Werten, auf die über eindeutige Schlüssel zugegriffen werden kann: Ein Wert (value) wird unter einem Schlüssel (key) abgelegt und später anhand dieses Schlüssels wiedergefunden. Beim Eintragen werden stets Paare aus Schlüssel und Wert angegeben. Ist dem Schlüssel noch kein Wert zugeordnet, wird das Paar aus Schlüssel und Wert zu der Map hinzugefügt. Ist der Schlüssel bereits vorhanden, dann wird der zugehörige alte Wert mit dem neuen Wert überschrieben. Beim Löschen wird nach dem Schlüssel gesucht und das Paar gelöscht, welches ihn enthält. Wird der Schlüssel nicht gefunden, passiert nichts.

Die Map soll als binärer Suchbaum implementiert werden, um das das Finden und Löschen von Elementen performant zu gestalten. Allerdings wird in dieser Aufgabe darauf verzichtet, die Bäume bei Bedarf auszubalancieren.

Auf die bereits existierenden Implementierungen von `Map` im Paket `scala.collection` soll in dieser Aufgabe *nicht* zurückgegriffen werden.

Folgende Typdefinitionen sind gegeben:

```
1 //Map Als Suchbaum
2 class MapEntry[K, V] (
3   var key : K,
4   var value : V,
5   var left : MapEntry[K, V] = null,
6   var right : MapEntry[K, V] = null)
7
8 class TreeMap[K, V] (var entries : MapEntry[K, V] = null)
```

a) Vervollständigen Sie die Implementierung folgender Funktionen und Prozeduren:

```
1  //leere map erstellen
2  def createMap[K, V]: TreeMap[K, V] = ...
3
4  //Anzahl der Elemente ermitteln
5  def size(map: TreeMap[_, _]): Int = ...
6
7  //Element fuer Schluessel key zurueckgeben sofern vorhanden
8  def get[K, V](less: (K, K) => Boolean)
9    (m: TreeMap[K, V], key: K) : V = ...
10
11 //Prueft, ob Schluessel key in m enthalten ist
12 def contains[K](less: (K, K) => Boolean)
13   (m: TreeMap[K, _], key: K) : Boolean
```

Die Funktion `get` soll eine `IllegalArgumentException` werfen, wenn der übergebene Schlüssel nicht gefunden wird. Die Funktion `contains` soll sich auf `get` abstützen.

Hinweis: Die übergebene Funktion `less` soll jeweils genau dann `true` zurückliefern, wenn der erste übergebene Schlüssel kleiner ist als der zweite.

> Folgende Implementierungen bieten sich für die Funktionen `createMap` und `size` an:
>
> ```
> 1 def createMap[K, V]: TreeMap[K, V] = new TreeMap[K, V]
> 2
> 3 def size(map: TreeMap[_, _]): Int = {
> 4   def s(entries: MapEntry[_, _]): Int = {
> 5     if (entries == null) return 0
> 6     else return s(entries.left) + 1 + s(entries.right)
> 7   }
> ```

```
  return s(map.entries)
}
```

Bei der folgenden Implementierung von `get` ist ein iteratives Vorgehen verfolgt worden:

```
def get[K, V](less: (K, K) => Boolean)(m: TreeMap[K, V], key: K)
    : V = {
  var p = m.entries
  while (p != null) {
    if (less(key, p.key)) p = p.left
    else if (less(p.key, key)) p = p.right
    else return p.value
  }
  //Kein Wert vorhanden
  throw new IllegalArgumentException("Element not found!")
}
```

Da `contains` hierbei auf `get` aufbauen soll, ist entsprechendes Exception-Handling notwendig:

```
def containsUsingGet[K](less: (K, K) => Boolean)(m: TreeMap[K, _
    ], key: K) : Boolean = {
  try {
    get(less)(m, key)
    return true
  }
  catch {
    case _ : IllegalArgumentException => return false
  }
}
```

b) Implementieren Sie eine Prozedur `printTreeMap(m: TreeMap[_, _])` die eine Map in folgendem beispielhaften Format auf der Konsole ausgibt:

```
5 ->FUENF
  1 -> EINS
    0 -> NULL
    3 -> DREI
  7 -> SIEBEN
    6 -> SECHS
    10 -> ZEHN
      9 -> NEUN
```

Hierbei beitet sich folgender rekursiver Ansatz an:

```
def printTreeMap(m: TreeMap[_, _]) : Unit = {
  def ptm(e: MapEntry[_, _], indent: String = ""): Unit = {
    if (e != null) {
      println(indent + e.key + " -> " + e.value)
      ptm(e.left, indent + "  ")
      ptm(e.right, indent + "  ")
    }
  }
  return ptm(m.entries)
}
```

c) Implementieren Sie eine Prozedur (!) `put`, die ein Paar aus Schlüssel und Wert in die Map einfügt. Bedenken Sie, dass der Schlüssel bereits in der Map vorhanden sein kann und reagieren Sie entsprechend.

Hier wird eine rekursive Umsetzung verwendet:

```
def put[K, V](less: (K, K) => Boolean)(m: TreeMap[K, V], key: K,
    value : V) : Unit = {
  //Rekursive Hilfsfunktion
  def p(entry : MapEntry[K, V], key : K, value : V) : MapEntry[K,
      V] = {
    if (entry == null) return new MapEntry[K, V](key, value)
    else if (less(key, entry.key)) {
      entry.left = p(entry.left, key, value)
    }
    else if (less(entry.key, key)) {
      entry.right = p(entry.right, key, value)
    }
    else entry.value = value

    return entry
  }
  m.entries = p(m.entries, key, value)
}
```

d) Legen Sie mithilfe der Funktion `createMap` und der Prozedur `put` eine Beispiel-Map (mit mindestens 10 Elementen) an, in der Städtenamen (Werte) ihren Postleitzahlen (Schlüssel) zugeordnet sind. Überlegen Sie sich zuvor geeignete Typen für Schlüssel und Werte und implementieren Sie eine entsprechende Funktion `less`.

Hinweis: Es bietet sich an, vorab eine geeignete Form von `put` durch Curryisierung zu definieren, bei der die Funktion `less` bereits berücksichtigt ist.

Für Postleitzahlen lässt sich der Typ `Int`, für Städtenamen sinnvollerweise der Typ `String` verwenden. Die nötige Funktion `iless` sowie die Hilfsfunktionen `iput` und `iget` lassen sich dann wie folgt definieren:

```
def iless : (Int, Int) => Boolean  = _ < _

def iput = put[Int, String](iless) _
def iget = get[Int, String](iless) _
```

Eine Auswahl an Städten lässt sich dann wie folgt eintragen:

```
val mc = createMap[Int, String]
iput(mc, 85521, "Ottobrunn")
iput(mc, 85579, "Neubiberg")
iput(mc, 82256, "Fuerstenfeldbruck")
iput(mc, 89073, "Ulm")
iput(mc, 99089, "Erfurt")
iput(mc, 44309, "Dortmund")
iput(mc, 20095, "Hamburg")
iput(mc, 14467, "Potsdam")
iput(mc, 24103, "Kiel")
iput(mc, 28777, "Bremen")
```

e) Implementieren Sie eine Prozedur

```
putAll[K, V](less : (K, K) => Boolean)
            (m1 : TreeMap[K, _], m2 : TreeMap[K, _]) : Unit = ...
```

die alle Elemente aus `m2` in `m1` überträgt und dabei bereits vorhandene überschreibt.

Auf Grundlage der Prozedur `put` ist folgender Ansatz (mit Hilfsprozedur `pa`) denkbar:

```
1  def putAll[K, V](less: (K, K) => Boolean)
2    (m1 : TreeMap[K, V], m2 : TreeMap[K, V]) : Unit = {
3    def pa(e : MapEntry[K, V]) : Unit = {
4      if (e != null) {
5        put[K, V](less)(m1, e.key, e.value)
6        pa(e.left)
7        pa(e.right)
8      }
9    }
10   return pa(m2.entries)
11 }
```

f) Wie viele rekursiven Aufrufe sind im besten bzw. schlechtesten Fall zum Suchen, Hinzufügen oder Entfernen eines Schlüssels in der Map der Größe n notwendig? Begründen Sie Ihre Aussage.

Gibt es Fälle, in denen dieser binäre Suchbaum kaum Vorteile gegenüber einer ungeordneten, verketteten Listenstruktur hat?

Im besten Fall (Element ist Wurzel des Baumes) ist nur 1 Schritt, im schlechtesten Fall (unbalancierter, einseitiger Baum) sind n Schritte notwendig. Wird eine bereits sortierte Folge von Schlüsseln eingefügt, ergibt sich kein Vorteil, da der Baum nur einseitig wächst. Je nach Zugriffverhalten kann die Liste sogar schneller sein.

Beispiel:

```
1  iput(map, 0, "NULL")
2  iput(map, 1, "EINS")
3  iput(map, 3, "DREI")
4  iput(map, 5, "FUENF")
5  iput(map, 6, "SECHS")
6  iput(map, 7, "SIEBEN")
7  iput(map, 9, "NEUN")
8  iput(map, 10, "ZEHN")
```

liefert

```
0 -> NULL
  1 -> EINS
    3 -> DREI
      5 -> FUENF
        6 -> SECHS
          7 -> SIEBEN
            9 -> NEUN
              10 -> ZEHN
```

Im Falle eines balancierten Baumes wäre die maximale Anzahl der Schritte $log_2(n)$.

g) Implementieren Sie eine Prozedur `def measure(action: =>Unit): Unit`, die die Ausführungszeit für eine beliebige Aktion misst und ausgibt. Um die Ausführungszeit zu messen, können Sie `System.nanoTime` verwenden um die aktuelle Zeit in Nanosekunden ausgibt. Damit können Sie sehr genau die Zeit vor und nach der Aktion bestimmen und durch die Differenz die Ausführungszeit.

Durch die Verwendung von Call-By-Name, wird die Übergebene `action` nicht beim Aufruf der Funktion ausgewertet, sondern erst beim Zugriff darauf. Wenn Call-by-Value verwendet worden wäre, wäre die Ausführungszeit nahe 0, da die Aktion schon beim Aufruf der Funktion und nicht erst zwischen den Messpunkten durchgeführt worden wäre.

```
1 def measure(action: => Unit): Unit = {
2     println("Starting to measure time")
3     val startTime = System.nanoTime
4     action
5     val endTime = System.nanoTime
6     println("Operation took "+(endTime-startTime)+" ns")
7 }
```

h) Oft ist es notwendig, die Schlüssel einer Map zu ermitteln. Verwenden Sie folgende Typdefinitionen, um eine Funktion `getKeys` zu implementieren, welche alle Schlüssel in sortierter Reihenfolge zurückliefert:

```
1 //Liste von Schluesseln
2 class KeyElem[K](var key : K, var next : KeyElem[K] = null)
3 class KeyList[K](var elems : KeyElem[K] = null)
4
5 def getKeys[K](map : TreeMap[K, _]) : KeyList[K] = ...
```

Hinweis: Sie können auf Listenfunktionen vergangener Übungsblätter oder aus der Vorlesung zurückgreifen.

Da es sich bei der Map um einen binären Suchbaum handelt, wird genau dann eine aufsteigend sortierte Liste erstellt, wenn man den Baum in *In-Order* durchläuft und dabei die Elemente in eine Liste einfügt. Mit einer Adaptierung der bekannten Funktion `append` lässt sich dies in naheliegender Wiese implementieren. Eine perfomantere Lösung ist die folgende:

```
1  def getKeys[K](map : TreeMap[K, _]) : KeyList[K] = {
2    def gk(e : MapEntry[K, _]) : (KeyList[K], KeyElem[K]) = {
3      if (e == null) return (null, null)
4      else {
5        val (left, leftLast) = gk(e.left)
6        val (right, rightLast) = gk(e.right)
7
8        val rtn = new KeyList[K]
9        val inner = new KeyElem[K](e.key)
10
11       if (left != null) {
12         rtn.elems = left.elems
13         leftLast.next = inner
14       }
15       else rtn.elems = inner
16
17       var last = inner
18       if (right != null) {
19         last.next = right.elems
20         last = rightLast
21       }
22       return (rtn, last)
23     }
24   }
25   return gk(map.entries)._1
26 }
```

> Bei dieser Beispielimplementierung wird eine Hilfsfunktion verwendet, die zu einem gegebenen `MapEntry` nicht nur die Liste aller enthaltenen Schlüssel zurückgibt, sondern auch einen Verweis auf das letzte Element. Auf diese Weise ist der Anfügevorgang einfacher und performanter, da die Liste, an die angehängt werden soll, nicht jedesmal (erneut) komplett durchlaufen werden muss.