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.

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?

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

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

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

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

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.

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.

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.

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.

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
```

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.

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.

e) Implementieren Sie eine Prozedur

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

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

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?

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.

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.