Team: 16, Talal Tabia, Michel Brüger

Aufgabenaufteilung:

Beide Teammitglieder bearbeiten alle Aufgaben gemeinsam

Quellenangaben: www.learnyousomeerlang.com, www.stackoverflow.com, www.wikipedia.org

Bearbeitungszeitraum:

02.04.2017 - 11.04.2017 ca. 40 Stunden

Aktueller Stand:

- Skizze fertig
- Liste, Stack, Schlange, BTree fertig implementiert

Änderungen in der Skizze: <Vor dem Praktikum auszufüllen: Welche Änderungen sind bzgl. der Vorabskizze vorgenommen worden.>

Skizze:
4 Benötigte Module:

liste
stack
queue
bTree

ADT Liste:
Objektmengen:

- pos — Position eines Elementes innerhalb einer Liste (1..n)
- elem — Ein in einer Liste enthaltenes Element
- list — Eine Liste mit n Elementen (0..n)

Datenstruktur:

Die Liste wird als Tupel {} realisiert.
Die innere Struktur einer Liste ist so definiert, dass das erste Element in der Liste durch eine weitere Liste gefolgt wird oder durch eine leere Liste.
Das Ende einer Liste wird durch ein leeres Tupel gekennzeichnet → {}

Beispiel1: Liste mit drei Elementen (1,x,3) -> {1, {x, {3, {}}}}
Beispiel2: Liste mit drei Elementen (1,(a,b),3) → {1,{{a, {b,{}}}, {3,{}}}}
Beispiel3: {{{2}}}, {{}} sind keine Listen.

create: ∅ → list
Erzeugt eine neue leere Liste -> {}

isEmpty: list → bool

Prüft ob eine Liste keine Argumente enthält.
Wenn **list = {}** dann wird true zurückgegeben, sonst false.

equal: list x list → bool

Prüft zwei Listen auf strukturelle Gleichheit.
Wenn beide Liste die gleichen Elemente an der selben Position haben, dann wird true zurückgegeben, sonst false.

laenge: list → int

Berechnet die Länge einer Liste.
Eine leere Liste hat die Länge 0.

Interner Ablauf:
Initialisierung Counter = 0.

Die gegebene Liste wird durchgelaufen und bei jedem gefundenen Element wird ein Counter hochgezählt, bis man beim Element{} angekommen ist. Das Ergebnis ist dann die Länge.

insert: list x pos x elem → list

Fügt ein Element an einer bestimmten Position in eine Liste ein.

Vorbedingung:
0 < Pos >= Länge der gegebenen List +1
Sollte die Vorbedingung verletzt, wird die Liste unverändert zurückgegeben.

Wenn **Pos = 1**

Wird das gegebene Element an ersten Stelle der neue Liste hinzugefügt.
Sollte die gegebene Liste nicht leer sein wird sie ab Position 2 an die neue Liste angehängt.

Wenn **Pos > 1**

Wird die gegebene Liste durchgelaufen und jedes enthaltende Element wird dabei an einer neuen Liste kopiert, bis die gewünschten Position erreicht ist.
Das neue Element wird dann an dieser Position in der neuen Liste hinzugefügt.
Sollte der Rest der gegebenen Liste weitere Elemente enthalten, wird dieser an die neue Liste an die nächste Position angehängt.

delete: list x pos → list

Entfernt ein Element an angegebener Position aus der Liste.
Alle eventuell nachfolgenden Elemente werden um eine Position nach vorne verschoben.
Ist an gegebener Position kein Element vorhanden wird die Liste nicht verändert.
Wird versucht an einer Position ein Element zu löschen, an der kein Element vorhanden ist, so wird die Liste nicht verändert.

Die Liste wird durchlaufen bis zum Element vor dem Element mit übergebener Position. Dabei wird jedes Element in eine neue Liste kopiert. Danach werden die Elemente hinter dem Element mit übergebener Position ebenfalls kopiert.

find: list x elem → pos

Durchläuft die Liste bis das Element welches dem als Argument übergebenen Element entspricht gefunden wurde und gibt dann dessen Position zurück.
Sind mehrere Elemente entsprechenden Wertes enthalten, so wird die Position des ersten gefundenen zurückgegeben.
Wird kein entsprechendes Element gefunden wird **nil** zurückgegeben.

- Argument1: **list** in welcher nach **elem** gesucht werden soll
- Argument2: **elem** nach welchem in **list** gesucht werden soll
- Rückgabe: **pos** von **elem**, bzw. **nil** falls kein entsprechendes Element gefunden wird.

retrieve: list x pos → elem

Durchläuft die Liste bis zu der übergebenen Position und gibt das dort befindliche Element zurück.

- Argument1: **list** in welcher nach **elem** gesucht werden soll
- Argument2: **pos** an welcher in **list** nach **elem** gesucht werden soll
- Rückgabe: **elem** welches sich an **pos** in **list** befindet, bzw. **nil** falls **pos** nicht in **list** enthalten ist.

concat: list x list → list

Verbindet zwei Listen miteinander indem Liste 2 an Liste 1 angehängt wird.
Durchläuft Liste 1 und kopiert alle Elemente in eine neue Liste, anschließend wird Liste2 durchlaufen und deren Elemente werden ebenfalls der Reihe nach an das Ende der neuen Liste angehängt.

- Argument1: **list1** an welche **list2** angehängt wird.
- Argument2: **list2** welche an **list1** angehängt wird.
- Rückgabe: neue **list** welche aus den Elementen von **list1** und **list2** besteht

diffListe: list x list → list

Erzeugt Liste welche alle Elemente enthält die nur in Liste1 vorkommen und nicht in Liste2.

Liste1 wird durchlaufen, dabei wird für jedes Element überprüft ob es in Liste 2 vorhanden ist. Ist es nicht in Liste 2 vorhanden kommt es in eine neue Liste.

- Argument1: **Liste1**

- Argument2: **Liste2**
- Rückgabe: **Liste**

eoCount: list → {gerade Counter,ungerade Counter}

Zählt die Anzahl aller in der Länge geraden Listen und die Anzahl aller in der Länge ungeraden Listen, welche in der Liste enthalten sind, sowie auch die Längen der in den Listen enthaltenen Listen.

Eine leere Liste wird als Liste gerader Länge angesehen.

Interner Ablauf:

Die Liste wird durchlaufen, dabei werden alle geraden und ungeraden enthaltenen Listen gezählt. Dies muss auch bei allen in der Liste enthaltenen Listen und deren Unterlisten usw. durchgeführt werden.
Anschließend wird das Ergebnis der Anzahl von geraden und ungeraden Listen zurückgeben.

ADT Stack

Datenstruktur:

Der Stack wird als Liste {} realisiert.
Die innere Struktur eines Stack ist so definiert, dass das oberste Element des Stacks das erste Element in der Liste ist.
Das zuletzt eingefügte Element ist das oberste Element im Stack und ist auch das zuerst entnehmbare.
Beispiel1: Stack mit drei Elementen (1,x,3) -> {1, {x, {3, {}}}}
Beispiel2: Stack mit drei Elementen (1,(a,b),3) → {1,{{a, {b,{}}}, {3,{}}}}
Beispiel3: {{{2}}}, {{}} sind keine Listen.

Objektmengen:

- elem — Ein in einem Stack enthaltenes Element
- stack — Ein Stack mit n Elementen (0..n)

createS: ∅ → stack

Erzeugt einen neuen leeren Stack

- Argumente: **keine**
- Rückgabe: **leerer stack**

push: stack x elem → stack

Legt ein Element vorne auf dem Stack ab

- Argument1: **stack** auf dem das Element abgelegt werden soll
- Argument2: **elem** welches auf **stack** abgelegt werden soll
- Rückgabe: **stack** welcher das neue abgelegte **elem** enthält

pop: stack → stack

Entfernt das oberste Element aus dem Stack.
Ist der Stack leer wird er nicht verändert.

- Argument: **stack** aus welchem das oberste **elem** entfernt werden soll.
- Rückgabe: **stack** aus welchem das oberste **elem** entfernt wurde, bzw. **leerer** **stack** falls **stack** vorher auch schon leer war.

top: stack → elem

Gibt das oberste Element des Stacks zurück.
Ist der Stack leer wird NULL zurückgegeben.

- Argument: **stack** aus welchem das oberste **elem** zurückgegeben werden soll
- Rückgabe: **elem** welches zuoberst auf dem **stack** liegt.

isEmptyS: stack → bool

Prüft ob der Stack keine Elemente enthält.

- Argument: **stack** der auf Inhalt überprüft werden soll
- Rückgabe: **true** falls **stack** leer ist, **false** falls nicht.

equalS: stack x stack → bool

Prüft zwei Stacks auf strukturelle Gleichheit.

- Argument1: **stack1**
- Argument2: **stack2**
- Rückgabe: **true** wenn beide **stacks** strukturell gleich sind, **false** wenn nicht.

reverseS: stack → stack

Dreht den Stack um.

Jedes Element des Stack wird auf einem neuen Stack abgelegt.
Dabei dreht sich die Reihenfolge um.
Es wird eine neue interne Liste erzeugt,

- Argument1: **stack**
- Rückgabe: umgedrehter **stack**

ADT Schlange

Datenstruktur:

Die Queue wird mittels zweier Stacks realisiert.
Ein InStack nimmt die neuen Elemente der Queue auf.
Ein OutStack stellt die Elemente der Queue zur Ausgabe bereit.

Beispiel1: Queue mit drei Elementen
(1,x,3) → {{1, {x, {3, {}}}},{}

Beispiel2: Queue mit drei Elementen
{ {1,{}} , {x,{3,{}}} }
Beispiel2: Stack mit drei Elementen (1,(a,b),3) → {1,{{a, {b,{}}}, {3,{}}}}
Beispiel3: {{{2}}}, {{}} sind keine Listen.

Objektmengen:

- elem Ein in einer Queue enthaltenes Element
- queue Eine Queue mit n Elementen (0..n)
-

createQ: ∅ → queue
Erzeugt eine leere Queue.
- Vorgehensweise:
  - Argumente: **keine**
  - Rückgabe: **leere Queue**

front: quele → elem
Liefert das vorderste Element aus der Queue zurück (das oberste Element im Outstack)
Falls der Outstack leer ist, müssen die Elemente des Instacks auf den Outstack gestapelt werden, dabei dreht sich die Reihenfolge um.

- Argument1: **queue**
- Rückgabe: das vorderste **elem** aus **queue**

enqueue: queue x elem → queue
Fügt ein Element hinten in der Queue ein.
Das gegebene Element wird zuoberst auf den InStack gelegt.

- Argument1: **queue** in welche **elem** eingefügt wird.
- Argument2: **elem** welches in **queue** eingefügt wird.
- Rückgabe: **queue** inklusive **elem.**

dequeue: queue → queue
Entfernt das vorderste Element aus der Queue (das oberste Element im Outstack)
Falls der Outstack leer ist, müssen die Elemente des Instacks auf den Outstack gestapelt werden, dabei dreht sich die Reihenfolge um.

- Argument1: **queue** aus welcher das vorderste **elem** entfernt wird.
- Rückgabe: **queue** ohne **elem.**

IsEmptyQ: queue → bool
Prüft ob eine Queue keine Elemente enthält (InStack und OutStack sind leer)
- Argument1: **queue** welche auf Inhalt geprüft wird.
- Rückgabe: **true** falls **queue** leer ist, **false** falls nicht.

equalQ: queue x queue → bool

Prüft ob zwei Queues strukturell gleich sind.
Die vorderen Elemente der beiden Queues werden verglichen, anschließend werden die vorderen Elemente entfernt. Dann werden wieder die vorderen Elemente der reduzierten Queues verglichen. Das wird so lange wiederholt bis beide Queues leer sind.

- Argument1: **queue1**
- Argumen2: **queue2**
- Rückgabe: **true** falls beide **queues** strukturell gleich sind, **false** falls nicht.

ADT Btree

Objektmengen:

- btree ein binärer Baum
- elem: Die hinzuzufügende Zahl

Datenstruktur:

Leerer binärer Baum → {}

Der binäre Baum hat intern die folgende Struktur:
Knoten → **{Elem,Hoehe,LinkBTree,RechtBTree}**

LinkBTree ist ein binärer Bäume dessen Knotenelemente alle kleiner sind als sein Wurzelelement.
RechtBTree ist ein binärer Bäume dessen Knotenelemente kleiner sind als sein Wurzelelement.
Die Blätter des binären Baumes haben folgende Struktur:
{Elem,Höhe,{},{}}

initBT: ∅ → btree
Erzeugt und liefert einen neuen leeren Btree zurück.

isBT: btree → bool
Überprüft ob btree von korrekter syntaktischer Struktur ist und ob die Semantik korrekt ist.

Interner Ablauf:

Initialisierung:
Setze Counter = 1

Es werden alle Knoten das Baumes durchlaufen. Dabei wird jeder Knoten dahingehend überprüft, ob er den strukturellen Anforderung des BTree entspricht. Also ob die Elemente des linken Teilbaums kleiner sind als die Wurzel, die Elemente des rechten Teilbaums größer sind als die Wurzel und die Höhe korrekt ansteigt.

Schritt 1: Wenn **btree = {}** liefere true zurück.
Wenn nicht gehe zu 2.

Schritt 2: Wenn **Höhe von btree = Counter**
Erhöhe counter um 1 und gehe zu 3.
Wenn nicht liefere false zurück.

Schritt 3: Wenn **elem von btree > elem von der linkbtree Wurzel**
und **elem von btree < elem von rechtbtree Wurzel**
gehe zu 4.
Wenn nicht liefere false zurück.

Schritt 4: Wiederhole Schritt 1 bis Schritt 3 für linkbtree und rechtbtree

insertBT: btree x elem □ btree

Fügt ein neues Element in den BTree ein.
Bei leerem BTree wird das Element die neue Wurzel.
Bei nicht leerem BTree wird das Element entsprechend seiner Größe relativ zur Wurzel einsortiert (Element < Wurzel → linker Teilbaum, Element >= Wurzel → rechter Teilbaum

Interner Ablauf:

Vergleicht das übergebene Element mit der Wurzel des Baumes. Ist es größer wird es die Wurzel des rechten Teilbaums, ist es kleiner die des linken.
Ist schon ein rechter bzw. linker Teilbaum vorhanden wiederholt sich der Vorgang dort.

- Argument1: **btree** in den **elem** eingefügt werden soll.
- Argument2: **elem** welches in **btree** eingefügt werden soll.
- Rückgabe: **btree** inklusive **elem**

isEmptyBT: btree → bool

Prüft ob ein BTree keine Elemente enthält.
Falls btree = {} wird true zurückgeben.

equalBT: btree x btree → bool

Überprüft zwei BTrees auf strukturelle Gleichheit.
Die Bäume werden parallel durchlaufen und auf gleiche Höhe geprüft, sodass jeder Knoteninhalt einer erreichten Höhe vom ersten btree strukturell gleich mit dem Knoteninhalt des äquivalenten Knotens in gleicher Höhe im zweiten btree sein muss.