**Team**: 16, Talal Tabia, Michel Brüger

**Aufgabenaufteilung**:

Beide Teammitglieder bearbeiten alle Aufgaben gemeinsam

**Quellenangaben**: [www.learnyousomeerlang.com](http://www.learnyousomeerlang.com/), [www.stackoverflow.com](http://www.stackoverflow.com/), www.wikipedia.org

**Bearbeitungszeitraum**:

02.04.2017 - 11.04.2017 ca. 24 Stunden

**Aktueller Stand**:

* Skizze fertig
* Liste fertig implementiert
* Stack 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.

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.

* + - Vorgehensweise: delete mit um 1 verringerter **pos** auf Rest von **list** aufrufen.  
      Abbruchbediingung1: **pos** = 1, dann neue Liste aus Rest von **list** erzeugen.
    - Argument1: **list** aus der **elem** entfernt werden soll
    - Argument2: **pos** an welcher **elem** aus **list** entfernt werden soll
    - Rückgabe: **list** welche die aus der Entfernung von **elem** resultierenden   
       Änderungen enthält.

**find**: list x elem → pos  
 Sucht in der Liste nach einem Element welches dem als Argument übergebenen Element entspricht.  
 Ist so ein Element enthalten wird dessen Position zurückgegeben.  
 Sind mehrere Elemente entsprechenden Wertes enthalten, so wird die Position des ersten gefundenen zurückgegeben.  
 Wird kein entsprechendes Element gefunden wird NULL zurückgegeben.

* + Vorgehensweise: **elem** mit Kopf von **list** vergleichen, wenn nicht gleich, dann find auf Rest von **list** aufrufen und Zähler um 1 erhöhen.  
    Abbruchbedingung1: Kopf von **list** = **elem**, dann Zähler zurückgeben.
    - Argument1: **list** in welcher nach **elem** gesucht werden soll
    - Argument2: **elem** nach welchem in **list** gesucht werden soll
    - Rückgabe: **pos** von **elem**, bzw. NULL falls kein entsprechendes   
       Element gefunden wird.

**retrieve**: list x pos → elem  
 Sucht nach einem Element an gegebener Position in der Liste.  
 Ist an entsprechender Position kein Element vorhanden wird NULL zurückgegeben.

* + Vorgehensweise: retrieve auf Rest von **list** aufrufen mit um 1 verringerter **pos**  
    Abbruchbedingung1: **list** ist leer, dann null zurückgeben.  
    Abbruchbedingung2: **pos** = 1, dann Kopf von **list** zurückgeben
    - 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.   
       NULLfalls **pos** nicht in **list** enthalten ist.

**concat**: list x list → list  
 Verbindet zwei Listen miteinander indem Liste 2 an Liste 1 angehängt wird.

* + Vorgehensweise: Liste erzeugen mit Kopf von **list1** als Kopf und Aufruf von concat mit Rest von **list1** und **list2** als Argumenten als Rest.  
    Abbruchbedingung1: **list1** ist leer, dann **list2** zurückgeben  
    Abbruchbedingung2: **list2** ist leer, dann **list1** zurückgeben
    - 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.

* + Vorgehensweise: Kopf von **list1** in **list2** suchen (verwendete Funktion: **find**); falls erfolgreich: diffListe mit Rest von **list1** und **list2** als Argumenten aufrufen, falls nicht erfolgreich: Liste erzeugen mit Kopf von **list1** als Kopf und Aufruf von diffList mit Rest von **list1** und **list2** als Argumenten.  
    Abbruchbedingung1: **list1** ist leer, dann leere Liste zurückgeben.
    - 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:

1. Berechne die Länge der Liste. Gehe zu 2.

2. Wenn die Länge gerade ist, dann wird der gerade Counter erhöht sonst der ungerade Counter. Gehe zu 3

3. Durchlaufe die Liste.

Wenn die Liste weitere Listen enthält, für jede enthaltene Liste gehe zu 1 .

4. Gebe {gerade Counter,ungerade Counter} zurück.

**ADT Stack**

Objektmengen:

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

**createS**: ∅ → stack  
 Erzeugt einen leeren Stack

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

**push**: stack x elem → stack  
 Legt ein Element auf dem Stack ab

* + Vorgehensweise:
    - 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.

* + Vorgehensweise:
    - 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.

* + Vorgehensweise:
    - 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.

* + Vorgehensweise:
    - 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.

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

**reverseS**: stack → stack

Dreht den Stack um.

* + Vorgehensweise:
    - Argument1: **stack**
    - Rückgabe: umgedrehter **stack**

**ADT Schlange**

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.

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

**enqueue**: queue x elem → queue  
 Fügt ein Element hinten in der Queue ein.

* + Vorgehensweise:
    - 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.

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

**IsEmptyQ**: queue → bool

Prüft ob eine Queue keine Elemente enthält.

* + Vorgehensweise:
    - 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.

* + Vorgehensweise:
    - 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 einen leeren BTree.

* + Vorgehensweise:
    - Argumente: **keine**
    - Rückgabe: leerer **btree**

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

Interner Ablauf:

Initialisierung:

Setze Counter = 1

**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**

und **isBT (linkbtree) = true und isBt (rechtbtree) = true**

**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

* + Vorgehensweise:
    - 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.

* + Vorgehensweise:
    - Argument1: **btree** dessen Inhalt überprüft wird.
    - Rückgabe: **true** falls **btree** keine Elemente enthält, **false** falls Elemente enthalten sind.

**equalBT**: btree x btree → bool  
 Überprüft zwei BTrees auf strukturelle Gleichheit.

* + Vorgehensweise:
    - Argument1: **btree1**
    - Argument2: **btree2**
    - Rückgabe: **true** falls beide **btrees** strukturell gleich sind, **false** falls nicht.