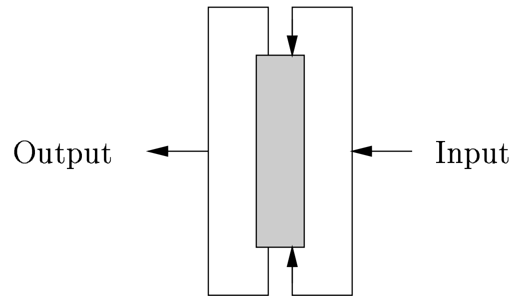
Aufgabe 1 $2 + 4 = 6T$

Abstrakte Datentypen

a) Gegeben sei ein ADT Dschlange mit Elementen aus der Menge Element, der ähnlich wie eine Schlange aufgebaut ist. Als Verallgemeinerung ist aber an beiden Enden sowohl das Einfügen als auch das Ausfügen von Elementen erlaubt.

Eine graphische Veranschaulichung des ADT's mit den erlaubten Operationen finden Sie in der nachfolgenden Abbildung:



Geben Sie für die folgenden Operationen die Signatur und die Axiome an:

`LeereDschlange` : Konstante für die leere Dschlange
`ausfügenRechts` und `ausfügenLinks` : löschen des Elements rechts bzw. links
`EinfügenRechts` und `EinfügenLinks` : anfügen eines Elements rechts bzw. links
`linkes` und `rechtes` : ermitteln des Elements ganz links bzw. ganz rechts
`dschlangeLeer` : Prüfung ob die Dschlange leer ist

Sie können Haskell-Notation verwenden.

b) Zeigen Sie, daß jeder Term der Form

`linkes(t),`

mit `t` Term vom Typ `Dschlange`, durch die Axiome in eine Normalform x mit x∈Element überführbar ist. Die Normalform ist dann genau dadurch gekennzeichnet, das in ihr keine Operationen der Schlange mehr vorkommen. Z.B. ist `y` die Normalform von (in Term-Notation)

`linkes(EinfügenRechts (EinfügenLinks(LeereDschlange,y),x)`

Lösung zu Aufgabe 1

ADT Doppelschlange:

a)
```
-- Elementtyp : t
data Dschlange t = LeereDschlange |
                   EinfuegenLinks  (Dschlange t) t |
                   EinfuegenRechts (Dschlange t) t

--Testdaten:
s1 :: Dschlange Int
s1 = (EinfuegenRechts LeereDschlange 1)
s2 :: Dschlange Int
s2 = EinfuegenRechts s1 2
s3 :: Dschlange Int
s3 = EinfuegenLinks s2 3
```

```
--  LeereDschlange     :: Dschlange
--  EinfuegenRechts    :: Dschlange -> Element -> Dschlange
--  ausfuegenRechts    :: Dschlange -> Dschlange
--  rechtes            :: Dschlange -> Element
--  EinfuegenLinks     :: Dschlange -> Element -> Dschlange
--  ausfuegenLinks     :: Dschlange -> Dschlange
--  linkes             :: Dschlange -> Element
--  dschlangeLeer      :: Dschlange -> Bool

-- Axiome:
ausfuegenRechts (EinfuegenLinks s e)
                |dschlangeLeer s      = s
                |otherwise            = EinfuegenLinks (ausfuegenRechts s) e
ausfuegenRechts (EinfuegenRechts s e) = s
ausfuegenRechts LeereDschlange        = error "Fehler"

ausfuegenLinks  (EinfuegenRechts s e)
                |dschlangeLeer s     = s
                |otherwise           = EinfuegenRechts (ausfuegenLinks s) e
ausfuegenLinks  (EinfuegenLinks s e) = s
ausfuegenLinks  LeereDschlange       = error "Fehler"

rechtes         (EinfuegenLinks s e)  |dschlangeLeer s = e
                                      |otherwise       = rechtes s
rechtes         (EinfuegenRechts s e)                  = e
rechtes         LeereDschlange                         = error "Fehler"

linkes          (EinfuegenRechts s e)  |dschlangeLeer s = e
                                       |otherwise       = linkes s
linkes          (EinfuegenLinks s e)                    = e
linkes          LeereDschlange                          = error "Fehler"

dschlangeLeer   (EinfuegenLinks s e)  = False
dschlangeLeer   (EinfuegenRechts s e) = False
dschlangeLeer   LeereDschlange        = True
```

b) Die Idee der Lösung ist, eine 'Bewertungsfunktion' $f : Dschlange \to \mathbb{N}_0$ für Terme zu finden, derart, daß $f(t) = 0$ genau dann, wenn t keine Operationen des ADT mehr enthält. Dann muß man noch zeigen, das jeder (korrekte) Term der Form $t = linkes(t')$ mit den Axiomen in einen Term t'' umgeformt werden kann mit $f(t) > f(t'')$. Hierfür bietet sich Induktion über den Aufbau von t' oder t (je nach gewählter Axiomatisierung) an. In obiger Axiomatisierung würde man $f(t)$ als Anzahl der ADT-Operationen in t wählen. Wenn man sich dann die Axiome für `linkes` anschaut, sieht man das diese Anzahl in allen Fällen kleiner gemacht werden kann. Wichtig ist dabei der Hinweis, daß `dschlangeLeer` immer in endlich vielen (hier einer) Schritten ausgewertet werden kann.

Aufgabe 2 **?**

Datenstrukturen

Unter einem Stapel (auch Keller genannt) versteht man eine Datenstruktur, auf die nur mit eingeschränkten Ein- und Ausfügeoperationen zugegriffen werden kann. Die Einfügeoperation, die mit dem Namen PUSH bezeichnet wird, legt jeweils ein Element der Eingabefolge als oberstes Element auf dem Stapel ab. Die Ausfügeoperation POP entfernt das oberste Element des Stapels.

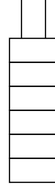
Wir betrachten nun die Permutationen, die aus der Folge $1, 2, ..., n$ erzeugt werden können.

Zeigen Sie, daß die Anzahl dieser Permutationen gleich

$$\binom{2n}{n} - \binom{2n}{n-1}$$

ist.

Ist z.B. $n = 2$, so wird die Permutation $1, 2$ durch die Operationsfolge PUSH,POP,PUSH,POP erzeugt. Die Permutation $2, 1$ wird durch PUSH,PUSH,POP,POP erzeugt. Die Operationsfolge PUSH,POP,POP,PUSH ist beispielsweise unzulässig, erzeugt also auch keine Permutation.



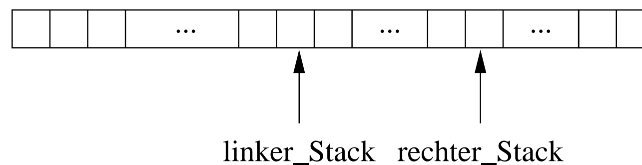
Lösung zu Aufgabe 2
Hier sollte man zuerst erkennen, das es genauso viele PUSH/POP-Folgen wie erzeugte Permutationen gibt. Das ist relativ offensichtlich, ein Beweis ist nicht unbedingt notwendig, der Hinweis auf diesen Sachverhalt reicht aus.

Dann stellt man die Anzahl der PUSH/POP-Folgen insgesamt auf, und erhält $\binom{2n}{n}$. Davon zieht man die Anzahl der nicht erlaubten (POP zu einem Zeitpunkt, zu dem der Keller leer ist) ab.

Aufgabe 3 **?**

Datenstrukturen Zeigen Sie, wie man zwei Stacks mit nur einer Reihung der Größe n implementieren kann, so daß zum einen die *push* und *pop* Operationen mit Aufwand $O(1)$ realisiert werden können und zum anderen keiner der beiden Stacks überläuft, es sei denn, daß die Gesamtzahl der Elemente der beiden Stacks größer als n wird.

Lösung zu Aufgabe 3



Die beiden Stacks wachsen jeweils von den Grenzen der Reihung zur Mitte hin. Der linke Stack wächst vom Reihungselement mit dem kleinsten Index in Richtung des Reihungselements mit höchstem Index. Der rechte Stack wächst vom Reihungselement mit dem größtem Index in Richtung des Reihungselements mit dem kleinsten Index.

Jeder Stack hat einen Zeiger, der auf das oberste Element des Stacks zeigt.

pop: als Wert wird der Wert des Reihungselements zurückgegeben, auf das der Stackzeiger des betreffenden Stacks zeigt. Der Zeiger verändert anschließend seine Position. Wurde pop auf den linken Stack angewendet, dann zeigt der Stackzeiger anschließend auf das Element links von dem, dessen Wert gelesen wurde. Wurde pop auf den rechten Stack angewendet, dann zeigt der Stackzeiger anschließend auf das Element rechts von dem, dessen Wert gelesen wurde.

push: ist zwischen beiden Stackzeigern kein Element mehr, dann Fehlersituation Überlauf. Sonst:

Soll das Element in den linken Stack eingefügt werden, dann verschiebe den linken Stackzeiger auf das nächste rechte Reihungselement und trage in das Reihungselement den Wert ein. Analog dazu push für den rechten Stack.

Aufgabe 4 **?**

Hashen Im Gegensatz zum verketteten Hashen, bei dem kollidierende Werte in Listen eingefügt werden, werden beim linear verschobenen Hashen die Schlüssel direkt in der Reihung abgespeichert. Für einen Schlüssel k wird dabei überprüft, ob an der Stelle $h(k)$ bereits ein Schlüssel steht. Ist dies der Fall, so werden die Stellen $h(k) + c, h(k) + 2c, ...$ untersucht, wobei c eine Konstante ist (beispielsweise $c = 1$).

Gegeben sei nun eine Reihung der Länge 13. Die Schlüssel

5, 1, 19, 23, 14, 17, 32, 30, 2

sollen in die anfangs leere Reihung eingetragen werden. Dabei sei $c = 1$ und $h(k) = k \ mod \ 13$.

a) Geben Sie die Reihungen für das verkettete Hashen und das linear verschobene Hashen an.

b) Geben Sie die Anzahl der beim Einfügen betrachteten Hashtabellenplätze für beide Verfahren an.

c) Welche Kosten sind für eine erfolgreiche Suche zu erwarten, wenn nach jedem Datum mit gleicher Wahrscheinlichkeit gesucht wird?

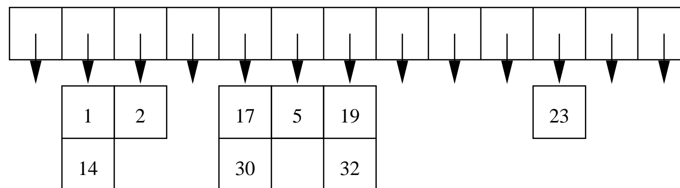
   Die Kosten seien hierbei die Anzahl der Lesezugriffe auf die Reihung.

Lösung zu Aufgabe 4

a) **1. linear verschobenes Hashen:**

-	1	14	2	17	5	19	32	30	-	23	-	-

2. verkettetes Hashen:



b) **1. linear verschobenes Hashen**: 16
2. verkettetes Hashen: 9

c) **1. linear verschobenes Hashen**: $\frac{16}{9}$
2. verkettetes Hashen: $\frac{12}{9} = \frac{4}{3}$

d) In der Praxis ist meist eine sehr große Schlüsselmenge K gegeben, z.B. die Menge aller Namen, die aus höchstens 20 Buchstaben bestehen. Von diesen Namen erwartet man eine bestimmte Anzahl a, z.B. 300.000.

Bei einer Implementierung als Bitvektor müßte man einen Vektor der Größe $O(|Alphabet|^{20})$ anlegen. Hier kann dann jeder Schlüssel direkt aufgefunden werden und bei der Suche würde ein Aufwand von $O(1)$ anfallen (Einfügen ebenfalls $O(1)$). Doch ist ein Array dieser Größe kaum noch zu realisieren. Und für die vergleichsweise wenigen Schlüssel auch etwas übertrieben ...

Beim 2. Verfahren bleibt nur die Möglichkeit, die Schlüssel 'wie sie kommen' zu speichern (Einfügeaufwand $O(1)$), wobei aber ein Suchaufwand von $O(n)$ anfällt. Selbst mittels einer Sortierung (Einfügeaufwand dann $O(n)$) kann man nur einen Suchaufwand von $O(\log n)$ erreichen (Intervallhalbierung, bei 300.000 Elementen ca. 18 Schritte).

Bei 3. und 4. müssen die Elemente dynamisch angelegt und freigegeben werden. Dies belastet die Speicherverwaltung u.U. erheblich. Bei 3. ergibt sich ein Einfügeaufwand von $O(1)$ und ein Such-/Löschaufwand von $O(n)$. Bei 4. ein Einfüge-, Such- und Löschaufwand von $O(\log n)$.

Findet man hingegen eine Hashfunktion, die schnell zu berechnen ist und die Schlüssel möglichst gleichverteilt auf die Tabelle verteilt, kann man den Suchaufwand trotz 'kleiner' Tabelle auf $O(1)$ reduzieren. Dabei darf der Aufwand für die Hashfunktion allerdings nicht schlechter sein, als derjenige für die $O(\log n)$-Verfahren. In der Praxis ist das oft erreichbar.

Theoretische Untersuchungen und praktische Messungen haben ergeben, daß die Tabelle nur zu 80% gefüllt werden sollte. Dann muß man im Durchschnitt mit 3 Kollisionen rechnen.

Quelle : Duden Informatik.

e) Der schlechteste Fall tritt ein, wenn die Hashfunktion nur eine Äquivalenzklasse liefert. Bei beiden Verfahren müssen im schlimmsten Fall alle Schlüssel betrachtet werden.

Gestaltet man die Hashtabelle jedoch wie beim verketteten Hashen, aber statt mit Listen mit balancierten sortierten Bäumen (z.B. AVL-Bäumen), muß man im schlimmsten Fall $O(log\ n)$ Vergleiche machen.

Aufgabe 5 **?**

<u>Die Türme von Hanoi</u>

Gegeben seien drei Stapel A, B und C. Der Stapel A besteht aus n Scheiben verschiedener Größen, die von unten nach oben in absteigender Größe aufeinander liegen. Die beiden anderen Stapel sind leer. Ziel ist es, alle Scheiben auf den Stapel B zu transferieren, wobei der Stapel C als Ablage verwendet werden kann. Auf jedem der Stapel darf zu keiner Zeit eine größere Scheibe auf einer kleineren liegen.

a) Leiten Sie eine Rekurrenz für die minimale Anzahl der benötigten Schritte her.

b) Geben Sie eine geschlossene Form für den Aufwand $T(n)$ an. Beweisen Sie Ihre Behauptung.

Lösung zu Aufgabe 3

a) Für $n \in \mathbb{N}_0$ sei $T(n)$ die minimale Anzahl von Zügen, mit denen n Scheiben korrekt von einem Stapel zu einem anderen transferiert werden können.

Strategie: Transferiere die größte Scheibe durch

1. Transferiere mit $T(n-1)$ Zügen die $n-1$ kleineren Scheiben von A nach C.
2. Transferiere die größte Scheibe nach B.
3. Transferiere die $n-1$ kleineren Scheiben mit $T(n-1)$ Zügen nach B.

$\Longrightarrow T(n) \leq 2T(n-1)+1$ (es könnte eine bessere Strategie geben)

Es geht nicht besser, denn: irgendwann **muß** die größte Scheibe transferiert werden. Zu diesem Zeitpunkt müssen sich die $n-1$ kleineren Scheiben korrekt gestapelt auf Stapel C befinden. Dies erfordert mind. $T(n-1)$ Züge.

$\Longrightarrow$ Die Rekurrenz:

$$\begin{aligned} T(0) &= 0 \quad (n=0) \\ T(n) &= 2T(n-1)+1 \quad (n>0) \end{aligned}$$

b) Berechne die ersten Funktionswerte

n	1	2	3	4	5	6
$T(n)$	1	$3 = 2\cdot 1+1$	$7 = 2\cdot 3+1$	$15 = 2\cdot 7+1$	$31 = 2\cdot 15+1$	$63 = 2\cdot 31+1$

und vermute $T(n) = 2^n - 1$ $(n \geq 0)$.

Beweis: durch vollständige Induktion über $n \in \mathbb{N}_0$.

I.A.: $n=0$

$T(0) = 0 = 2^0 - 1$

I.H.: Es gibt ein $n \in \mathbb{N}_0$, so daß die Behauptung für alle $k \leq n$ gilt.

I.S.: $n \rightarrow n+1$

$T(n+1) = 2\cdot T(n-1)+1 \overset{\text{I.H.}}{=} 2\cdot(2^n-1)+1 = 2^{n+1}-1$