Till Mossakowski
Benjamin Junge

6. Excercise sheet

Issued: 2023-11-21
Due: 2023-11-27 & 2023-11-28

6.1 *Keeping things sorted*

Write a module `SortedList` with a corresponding type that behaves like the standard list type but which ensures the invariant that the list is sorted at all times. Provide operations on that type that are corresponding to `[]`, `(:)`,`head`, `tail`, `null` and `length` and that don't break the invariant. Encapsulate your type s.t. the invariant can't be destroyed by a user importing your module. Test your module accordingly.

6.2 *Deques*

Write a module `Deque` with corresponding type `Deque a` that implements a double ended queue (or deque for short). The module should provide the following functions on deques to the user:

```
pushFront :: a → Deque a → Deque a
pushEnd :: a → Deque a → Deque a
peekFront :: Deque a → Maybe a
peekEnd :: Deque a → Maybe a
popFront :: Deque a → Deque a
popEnd :: Deque a → Deque a
makeDequeFromList :: [a] → Deque a
```

The push operations insert an element in the front or end respectively. The pop operations remove an element in the front or end respectively. The peek operations return the element currently at the front/end position. Encapsulate your type accordingly.

6.3 *Deques as Stacks/Queues*

Based on 6.2 write the modules `MyStack` and `MyQueue` with corresponding type (and the usual operations on that type) that implement a Stack and a Queue based on your module/type Deque.

Estimate the worst case complexity of the operations of the modules Deque, MyStack and MyQueue in big-O-notation. Can the Stack and Queue type be implemented more efficiently? If so give a rough sketch of a possibly better implementation.

6.4 *Vectors*

Write a module `MVector` with type `Vector a` that implements the mathematical concept of vectors and provides the following standard operations on them:

```
vectorFromList :: Num a ⇒ [a] → Vector a
skalarMult :: Num a ⇒ a → Vector a → Vector a
vecAdd :: Num a ⇒ Vector a → Vector a → Maybe (Vector a)
vecProd :: Num a ⇒ Vector a → Vector a → Maybe a
crossProd :: Num a ⇒ Vector a → Vector a → Maybe (Vector a)
```

Ensure that only vectors of same length can be added/multiplied and the crossproduct can only be used with vectors of length 3.

6.5 * *Maps and a very clunky TicTacToe*

A game of Tic-Tac-Toe consists of 5 things: a Grid that contains the current state (placement of the tokens), the games status (e.g. "game hasn't finished yet", "player x won" etc.), a symbol[1] for player one, a symbol of player two and the number of the current turn.
For the Grid use the module `Data.Map`[2]. Choose a good key type and use String as the Value type. Based on this description of a possible type for a game of Tic-Tac-Toe write a module `TicTacToe` with the `Game` and `Grid` types that exports the functions:

```
newGame :: String → String → Game
doTurn :: Game → k → Game
doTurns :: Game → [k]→ Game
```

k is a placeholder for you keytype used in the Grid type
newGame creates a Game at turn 0 with the two tokens for player one and two provided as Strings and an empty Grid.
doTurn takes a Game and a key(k) and behaves as follows:

- if the current turn number is > 10 or the game status is already at "player x won" it returns the game with the information which player won (or if it is a draw) set as the game status
- if a token is already in place k then return the game with unaltered grid and game status "place already has a token"
- if k is a invalid value for the grid then return the game with unaltered grid and game status "invalid grid position"
- if none of the above held then: if the turn is even place the token of player one at position k, else place the token of player two at position k. in both cases increment the turn number by 1 and check if one of the players won, if so set the game status accordingly.

doTurns takes a list of keys and applies them in order as turns using doTurn. Write a custom Show instance for the Game type that displays somewhat like this:

```
Game hasn't finished yet
 |  | W
 ———————
W|W|  !
 ———————
 |  |
Turn: 6
```

Where a `newGame "W" "!"` got used to create the example output. Your show instance doesn't have to account for symbols with more than one Char.

[1] Symbol here doesn't refer to Char. A symbol for a token can be a String of any length
[2] For more informations see Data.Map