Principles of Functional Languages (PFL)

Unit-2

Abstract Data Types

Stack Implementation in Haskell

```
-- Stack data type
data Stack a = Stack [a]
  deriving Show

-- Push an element onto the stack
push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (x : xs)

-- Pop an element from the stack
pop :: Stack a -> Maybe (a, Stack a)
pop (Stack (x:xs)) = Just (x, Stack xs)
pop _ = Nothing  -- Return Nothing if the stack is empty

-- Peek at the top element of the stack
peek :: Stack a -> Maybe a
peek (Stack (x:_)) = Just x
peek _ = Nothing  -- Return Nothing if the stack is empty

-- Check if the stack is empty
isEmpty :: Stack a -> Bool
isEmpty (Stack xs) = null xs

-- Example usage:
main :: IO ()
main = do
    let stack1 = Stack [1, 2, 3]
    print stack1

    let stack2 = push 4 stack1
    print stack2  -- Stack [4, 1, 2, 3]

    let (top, stack3) = case pop stack2 of
                          Just (x, s) -> (x, s)
                          Nothing -> (0, stack2)
    print top       -- 4
    print stack3    -- Stack [1, 2, 3]

    print $ peek stack3  -- Just 3
    print $ isEmpty stack3  -- False
```

Queue Implementation

```
-- Queue data type
data Queue a = Queue [a] [a]
  deriving Show

-- Enqueue an element into the queue
enqueue :: a -> Queue a -> Queue a
enqueue x (Queue front back) = Queue front (x : back)

-- Dequeue an element from the queue
dequeue :: Queue a -> Maybe (a, Queue a)
dequeue (Queue (x:front) back) = Just (x, Queue front back)
dequeue (Queue [] back) = case reverse back of
                            (x:front) -> Just (x, Queue front [])
                            [] -> Nothing  -- Return Nothing if the
queue is empty

-- Peek at the front element of the queue
peekQueue :: Queue a -> Maybe a
peekQueue (Queue (x:_) _) = Just x
peekQueue _ = Nothing  -- Return Nothing if the queue is empty

-- Check if the queue is empty
isEmptyQueue :: Queue a -> Bool
isEmptyQueue (Queue [] []) = True
isEmptyQueue _ = False

-- Example usage:
mainQueue :: IO ()
mainQueue = do
    let queue1 = Queue [1, 2] [3, 4]
    print queue1

    let queue2 = enqueue 5 queue1
    print queue2  -- Queue [1,2] [3,4,5]

    let (front, queue3) = case dequeue queue2 of
                            Just (x, q) -> (x, q)
                            Nothing -> (0, queue2)
    print front    -- 1
    print queue3   -- Queue [2] [3,4,5]

    print $ peekQueue queue3  -- Just 2
    print $ isEmptyQueue queue3  -- False
```

Implementing queue using two lists

```
-- A more efficient Queue using two lists
data Queue a = Queue [a] [a] -- Front part and back part
  deriving Show

-- Enqueue operation (O(1))
enqueue :: a -> Queue a -> Queue a
enqueue x (Queue front back) = Queue front (x : back)

-- Dequeue operation (O(1) amortized)
dequeue :: Queue a -> Maybe (a, Queue a)
dequeue (Queue (x:front) back) = Just (x, Queue front back)
dequeue (Queue [] back) = case reverse back of
                            (x:front) -> Just (x, Queue front [])
                            [] -> Nothing  -- Return Nothing if the
queue is empty
```

Circular Queue Implementation

```
-- A simple Circular Queue implementation
data CircularQueue a = CircularQueue { front :: Int, rear :: Int, size
:: Int, queue :: [a] }
  deriving Show

-- Create a new CircularQueue with a given size
createQueue :: Int -> CircularQueue a
createQueue n = CircularQueue 0 0 n (replicate n undefined)

-- Enqueue operation (circular behavior)
enqueueCQ :: a -> CircularQueue a -> CircularQueue a
enqueueCQ x (CircularQueue f r size q) =
    if (r + 1) `mod` size == f then error "Queue is full"
    else let newQueue = take r q ++ [x] ++ drop r q
             newRear = (r + 1) `mod` size
         in CircularQueue f newRear size newQueue

-- Dequeue operation
dequeueCQ :: CircularQueue a -> (a, CircularQueue a)
dequeueCQ (CircularQueue f r size q) =
    if f == r then error "Queue is empty"
    else let x = q !! f
             newQueue = take f q ++ drop (f+1) q
             newFront = (f + 1) `mod` size
         in (x, CircularQueue newFront r size newQueue)
```

Priority Queue Implementation

```
-- Simple Priority Queue using a list
data PriorityQueue a = PriorityQueue [(Int, a)] -- (priority, value)
```

```
  deriving Show

-- Insert an element with a given priority
insertPQ :: Int -> a -> PriorityQueue a -> PriorityQueue a
insertPQ p x (PriorityQueue xs) = PriorityQueue ((p, x) : xs)

-- Remove the element with the highest priority
removePQ :: PriorityQueue a -> Maybe (a, PriorityQueue a)
removePQ (PriorityQueue []) = Nothing
removePQ (PriorityQueue xs) = Just (snd maxElement, PriorityQueue
rest)
  where
    maxElement = maximumBy (comparing fst) xs
    rest = filter ((/= fst maxElement) . fst) xs
```