```fsharp
module Poolean.HuffmanDomain

open NucleotideDomain
open CompressionDomain

module HuffmanDomain =

    type ASCIICount = { Key: char; Value: int} // could change for n-grams (string) and frequency (float)

    /// 1's based Heap implementation (Priority Queue)
    /// - Smallest (integer) items have the highest priority
    ///
    /// See Kleinberg & Tardos for more details (2nd edition, p. 60)
    /// - This implementation is not thread-safe
    type Heap() =
        let mutable size = 0
        let mutable queue = [| {Key = '\000'; Value = 0} |] // first index is a placeholder, don't use it

        let swap e1 e2 =
            if (queue.[e2]).Value < (queue.[e1]).Value then
                let tmp = queue.[e1]
                queue.[e1] <- queue.[e2]
                queue.[e2] <- tmp

        member self.Print() = printfn "Queue:\t%A\nSize:\t%d" queue size

        /// find parent index of node i
        member self.Parent(i) = floor((i |> float) / 2.0) |> int

        /// bubble values up the heap
        member self.HeapifyUp(i) =
            if i > 1 then
                let j = self.Parent(i)
                swap j i
                self.HeapifyUp(j) // more efficient if I checked i < j, this will go up the tree with NOPs

        /// bubble values down the heap
        member self.HeapifyDown(i) =
            let n = size
            match (2*i > n), (2 * i < n), (2 * i = n) with
            | _, true, _ ->
                let left = 2*i
                let right = 2*i + 1
                (match (queue.[left]).Value < (queue.[right]).Value with | true -> left | false -> right) |> Some
            | _, _, true -> 2*i |> Some
            | _, _, _ -> None
            |> Option.bind (fun j -> swap i j; self.HeapifyDown(j); Some j) |> ignore

        /// insert value into heap H
        /// use heapify-up to repair damaged heap structure after each call
        /// new elements get appended to the end of the internal array
        member self.Insert(v) =
            queue <- Array.append queue [|v|]
            size <- size + 1
            self.HeapifyUp(size)

        /// if heap contains elements, return minimum element
        member self.FindMin() = match size >= 1 with | true -> Some queue.[1] | false -> None

        /// Delete element in heap position i
        /// use heapify-down to repair damaged heap structure after each call
        member self.Delete(i) =
            queue <- Array.append queue.[0..i - 1]  queue.[i+1 .. size]
            size <- size - 1
            self.HeapifyDown(i)

        /// identify and delete element with minimum key value
        member self.ExtractMin() = self.FindMin() |> Option.bind (fun min -> self.Delete(1); Some min)
```