Assignment 4

Exercise 1

Create a library called MultiSet to represent ordered multisets in F#. Remember that a library contains both a signature file and an implementation file. A multiset is a set that can contain duplicate elements, so set's such as {x,y,x}\{x, y, x\}{x,y,x} are perfectly valid. The number of times an element occurs in a set is called the ***multiplicity*** of that element. An ordered multiset also allows you to retrieve the elements in order of some given comparison relation. You may use whatever data type you like to store your set, but by far the easiest (and it will give you all ordering properties completely for free) is `Map<'a, uint32> when 'a : comparison`, where the key of the map represents elements in the set and the values of the map represents the number of occurrences of their corresponding keys. So the example set before could be implemented using the map [x↦2u;y↦1u][x \mapsto 2u; y \mapsto 1u][x↦2u;y↦1u]. Other things you may consider including is the size of the set so that size lookups can be done in constant rather than linear time, but that is entirely optional.

A short recapitulation on unsigned integers is in order. A 32 bit unsigned integer has the type `uint32` and there literals have a `u` postfix (e.g. `42u`). Moreover F# does not directly allow operators of different types to, for instance, be added so the expression `5 + 8u` will not type check, but casting the expression to `5 + (int 8u)` is perfectly valid.

For the description below, we will use the notation `{(a1, x1), ..., (an, xn)}` for a multiset with elements `a1...an` where `xi` represents how many occurrences of `ai` are in the set. Moreover, the set is ordered so we know that for all iii and jjj, such that i<ji < ji<j, ai<ajai < ajai<aj.

The library should contain the following functions:

`empty : MultiSet<'a>` that creates an empty set.

`isEmpty : MultiSet<'a> -> bool` that given a set `s`, return `true` if a `s` is empty, and `false` otherwise.

`size : MultiSet<'a> -> uint32` that given a set `s`, return the size of `s`

`contains : 'a -> MultiSet<'a> -> bool` that given an element `a` and a set `s`, return `true` if `a` is an element of `s` and `false` otherwise.

`numItems : 'a -> MultiSet<'a> -> uint32` that given an element `a` and a set `s`, return how many occurrences of `a` there are in `s`.

`add : 'a -> uint32 -> MultiSet<'a> -> MultiSet<'a>` that given an element `a`, a number `n`, and a set `s`, return the set that has `a` added to `s n` times.

`addSingle : 'a -> MultiSet<'a> -> MultiSet<'a>` that given an element `a` and a set `s`, return `s` with a single nstance of `a` added to it.

`remove : 'a -> uint32 -> MultiSet<'a> -> MultiSet<'a>` that given an element `a`, a number `n`, and a set `s`, return the set with `n` occurrences of `a` from `s` removed (remember that a set cannot have fewer than 0 entries of an element). If `s` does not contain `n` occurrences of `a` then the result should contain no occurrences of `a`.

`removeSingle : 'a -> MultiSet<'a> -> MultiSet<'a>` that given an element `a` and a set `s` return the set where a single occurrence of `s` has been removed. If `s` does not contain `a` then return `s`.

`fold : ('a -> 'b -> uint32 -> 'a) -> 'a -> MultiSet<'b> -> 'a` that given a folding function `f`, an accumulator `acc` and a set `{(a1,x1), ..., (an,xn)}`, return `f (... (f acc a1 x1) ...) an xn)`. Note that the folder function `f` takes the number of occurrences of each element as an argument rather than calling `f` that number of times. It is up to the user of the `fold` method to determine if he wants that type of behaviour and code his folder function accordingly if so (this is better practice, and it makes your life easier).

`foldBack : ('a -> uint32 -> 'b -> 'b) -> MultiSet<'a> -> 'b -> 'b` that given a folding function `f`, an accumulator a set `{(a1,x1), ..., (an,xn)}`, and an accumulator `acc`, return `f a1 x1 (... (f an xn acc) ...)`. The same reasoning applies to `foldBack` as `fold`.

`map : ('a -> 'b) -> MultiSet<'a> -> MultiSet<'b>` that given a mapping function `f` and a set `{(a1, x1), ..., (an, xn)}` returns `{(f a1, x1), ..., (f an, xn)}`. Note that the mapping is only done on the element itself, not the number of times it occurs. This is in line with how maps generally work - they change the payload, but not the size of a collection. Also note that this one is actually a bit trickier than usual as you cannot use the regular Map.map to solve this, as that map leaves the keys unchanged, and you are most likely using your keys to represent the elements of your set.

`ofList : 'a list -> MultiSet<'a>` that given a list `lst` returns a set containing exactly the elements of `lst`

`toList : MultiSet<'a> -> 'a list` that given a set `{(a1, x1), ..., (an, xn)}` returns `[a1; ..(x1 times)..; a1; ...; an; ..(xn times)..; an]`

`union : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given sets `s1` and `s2` returns the union of `s1` and `s2`. The union of two multisets keeps the largest multiplicity in the result set (if an element occurs two times in `s1` and three times in `s2` then it will appear three times in the result).

`sum : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given sets `s1` and `s2` returns the sum of `s1` and `s2`. The sum of two multisets adds the multiplicities of all elements together in the result set (if an element occurs two times in `s1` and three times in `s2` then it will appear five times in the result).

`subtract : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given sets `s1` and `s2` subtracts `s2` from `s1`

`intersection : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given sets `s1` and `s2` returns the intersection of `s1` and `s2`

Finally, override the `ToString()` method of `MultiSet` such that a set `{(a1, x1), ..., (an, xn)}` is converted to the string string `{(a1, #x1), ..., (an, #xn)}` where the `ToString()` method for `'a` is used to convert `a1...an` to strings.

Exercise 2

Create a library called Dictionary to represent a dictionary of, for example, the English language in F#. Remember that a library contains both a signature file and an implementation file.

For the Scrabble assignment you will need to use a Trie (a prefix tree) to store your dictionary, and you are highly recommended to do so here as well. However, you are allowed to use simpler structures (like lists) if you wish, but these won't be nearly as fast and you will not be able to pass all the tests. You will, however, get full marks for a slower simpler solution. If, on the other hand, you want feedback on your dictionary then now is the best time to get started.

Your library must contain the following three functions:

`empty : string -> Dictionary` that given a string `s` containing all of the letters of the alphabet that we use in the dictionary, returns an empty dictionary that expects to be populated with words using only letters from this alphabet. The English language, for instance, would be instantiated by the call `empty "ABDEFGHIJKLMNOPQRSTUVWXYZ"`. You may assume that the string contains no duplicate characters and that the functions bellow only function correctly when fed words consisting only of letters in this alphabet. You may **not** assume that 'a' and 'A' are the same character (which makes your life much easier) as we will have weird languages in the future where this is not the case.

`insert : string -> Dictionary -> Dictionary` that given a string `s` and a dictionary `dict` adds the word `s` to `dict`. This function must not be (but can be) functional, i.e. you are allowed to use destructive updates within `dict` that actually changes the value of the argument passed to the function. This should be avoided when doing functional programming, but is permissible here for efficiency reasons.

`lookup : string -> Dictionary -> bool` that given a string `s` and a dictionary `dict` returns `true` if `s` is in `dict` and `false` otherwise.