Trabajo Práctico 1

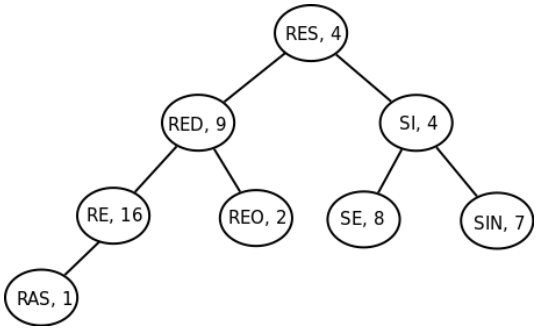
Un diccionario se define como un conjunto de pares (clave, valor) donde las claves son usada para encontrar rápidamente datos almacenados en los valores. Cada clave tiene asociado un único valor y debe ser comparable con las demás (es decir, debe existir un orden entre las claves). Las operaciones que deben estar en un diccionario son:

- crear un diccionario vacío
- insertar un elemento al diccionario
- eliminar un elemento al diccionario
- buscar el valor asociado a una clave en un diccionario
- listar el conjunto de claves de un diccionario

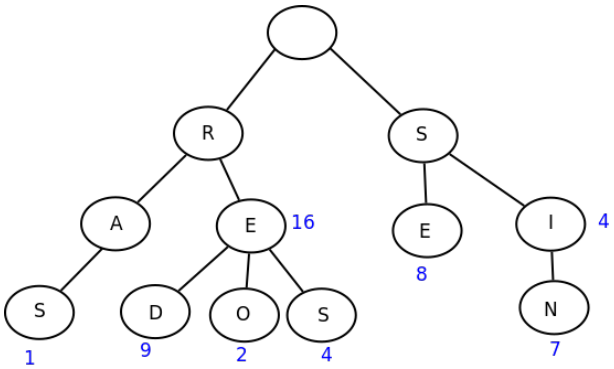
Veremos una estructura de datos que es utilizada para representar diccionarios cuyas claves son cadenas de caracteres. La forma en la que se almacena la información en esta estructura permite hacer búsquedas eficientes de cadenas que comparten prefijos, por esta razón es comúnmente usada para representar diccionarios en los teléfonos móviles, en los cuales los diccionarios son usados por ejemplo para el autocompletado.

La estructura que utilizaremos se obtiene al combinar: árboles binarios de búsqueda, los cuales proporcionan búsquedas eficientes sobre valores de cualquier tipo y otra clase de árboles usados específicamente para la búsqueda eficiente de cadenas que comparten prefijos.

En la figura 1 vemos un BST usado para representar el siguiente diccionario:
$\{(\texttt{"se"}, 8); (\texttt{"si"}, 4); (\texttt{"sin"}, 7); (\texttt{"ras"}, 1); (\texttt{"re"}, 16); (\texttt{"red"}, 9); (\texttt{"res"}, 4); (\texttt{"reo"}, 2)\}$

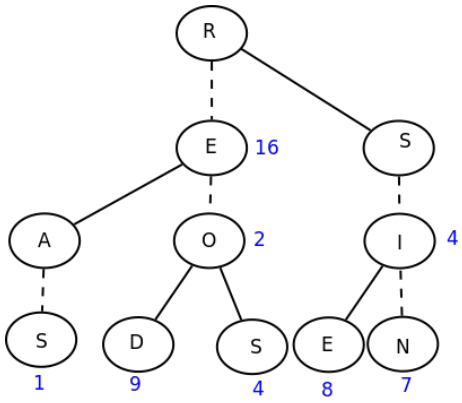


También podríamos representar el mismo diccionario mediante el siguiente árbol:



Una búsqueda en este tipo de árboles comienza por la raíz y continúa por las ramas de manera que el recorrido desde la raíz sea siempre un prefijo de la cadena a buscar.
Las búsquedas de cadenas que comparten prefijos son más rápidas usando esta última representación.

Las estructuras de datos anteriores pueden combinarse, obteniéndose un tipo de árbol de búsqueda, que también permite la búsqueda rápida de cadenas que comparten prefijos. Una búsqueda en este tipo de árboles compara el primer caracter de la palabra con la raíz del árbol, si éste caracter es menor, la búsqueda continúa por el subárbol izquierdo, si es mayor continúa por el subárbol derecho y si es igual por el subárbol del medio tomando el siguiente caracter de la palabra como nuevo caracter a buscar.
En la figura 3 vemos un árbol de este tipo usado para representar el mismo diccionario que representan los árboles anteriores:



Por ejemplo, si quisiéramos buscar el valor asociado a la cadena `"si"` es este árbol, comenzaríamos comparando el caracter `'s'` con la raíz del árbol, como éstos no coinciden la búsqueda continúa sobre el subárbol derecho (ya que `'s'` > `'r'`), la raíz de éste subárbol sí coincide con `'s'`, por lo tanto se busca la cadena `"i"` en el subárbol del medio, en el cual se encuentra que el valor asociado a `"si"` es 4.

Utilizaremos el siguiente tipo de datos en Haskell para representar árboles de búsqueda como los del ejemplo anterior con claves de tipo $[k]$ y valores de tipo v:

```
data TTree k v = Node k (Maybe v) (TTree k v) (TTree k v) (TTree k v)
               | Leaf k v
               | E
```

donde el valor de tipo Maybe v que guardan los nodos (o las hojas) está asociado a la clave que se forma al recorrer el camino que va desde la raíz al nodo (a la hoja). Además se debe cumplir el siguiente invariante para que éstos árboles sean de búsqueda: si un nodo almacena una clave x, las claves del subárbol izquierdo deben ser menores o iguales a x y las claves del subárbol derecho deben ser mayores a x.

Podemos representar el árbol del ejemplo con el siguiente valor t de tipo TTree Char Int:

```
t = Node 'r' Nothing E (Node 'e' (Just 16)  (Node 'a' Nothing E (Leaf 's' 1) E)
                                            (Node 'o' (Just 2) (Leaf 'd' 9)
                                                               E
                                                               (Leaf 's' 4))
                                            E)
                       (Node 's' Nothing E (Node 'i' (Just 4) (Leaf 'e' 8)
                                                              (Leaf 'n' 7)
                                                                E)
                                            E)
```

Definir las siguiente funciones en Haskell para manipular árboles de tipo TTree k v:

1. search :: Ord $k \Rightarrow [k] \rightarrow$ TTree k $v \rightarrow$ Maybe v, devuelve el valor asociado a una clave.
2. insert :: Ord $k \Rightarrow [k] \rightarrow v \rightarrow$ TTree k $v \rightarrow$ TTree k v, agrega un par (clave, valor) a un árbol. Si la clave ya está en el árbol, actualiza su valor.
3. delete :: Ord $k \Rightarrow [k] \rightarrow$ TTree k $v \rightarrow$ TTree k v, elimina una clave y el valor asociada a ésta en un árbol.

4. keys :: TTree $k\ v \to [[k]]$, dado un árbol devuelve una lista ordenada con las claves del mismo.

5. Se utilizará la siguiente clase en Haskell para representar diccionarios:

```
class Dic k v d | d → k v where
  vacio    :: d
  insertar :: Ord k ⇒ k → v → d → d
  buscar   :: Ord k ⇒ k → d → Maybe v
  eliminar :: Ord k ⇒ k → d → d
  claves   :: Ord k ⇒ d → [(k, v)]
```

   En la declaración de la clase Dic, la dependencia funcional $d \to k\ v$ dice que el tipo de k y v queda determinado por el tipo de d. Esta dependencia no cambia la sintaxis de la declaración de instancias usual.

   Para poder compilar este código se necesitan algunas extensiones del lenguaje Haskell, para ello agregar como encabezado del archivo en Haskell las siguientes líneas:

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
```

   Dar una instancia de la clase Dic para el tipo de datos TTree $k\ v$.