2. Enunciado de la práctica

La práctica consiste en elaborar un programa en **Haskell** que permita al usuario usar arrays dispersos y mostrar que se puede acceder y modificar sus elementos en un tiempo logarítmico con respecto al índice utilizado. Consideraremos que los índices son valores enteros mayores o iguales a cero.

Se implementarán tres operaciones para trabajar con los arrays dispersos:

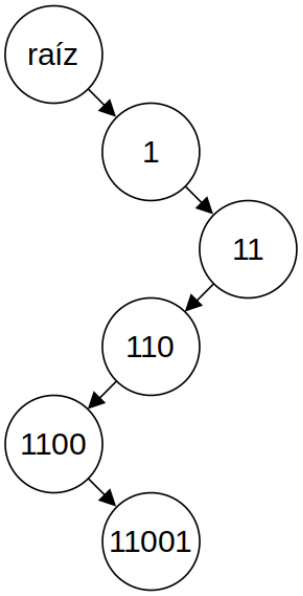
1. **`set`**: que recibirá un array disperso `a`, un índice `i` y un elemento `e` y devolverá el array disperso resultado de insertar en `a` el elemento `e` bajo el índice `i`. Si antes de la inserción hubiera un elemento en `a` indexado bajo ese mismo índice `i`, entonces el elemento anterior se perdería, siendo substituido por `e`.
2. **`get`**: que recibirá un array disperso `a` y un índice `i` y devolverá el elemento de `a` que esté indexado bajo el índice `i`. Si no existe, devolverá un valor `Null` (ver apartado de implementación).
3. **`delete`**: que recibirá un array disperso `a` y un índice `i` y devolverá el array disperso resultado de eliminar de `a` el elemento indexado bajo el índice `i`. Si no existiera un elemento indexado bajo ese índice, se devolverá el array disperso `a` sin ninguna modificación.

Estos arrays se van a implementar mediante un árbol binario en el que cada nodo (salvo la raíz) estará asociado a un número entero desde el 0 en adelante. Para localizar el nodo correspondiente a un número n, se realiza el siguiente proceso:

1. Comenzamos el recorrido en el nodo raíz del árbol binario.
2. Se recorre la secuencia de dígitos de la representación binaria de la posición buscada, desde el dígito más significativo al menos significativo.
   a. Si el dígito es un 0, se desciende por el hijo izquierdo del nodo actual.
   b. Si el dígito es un 1, se desciende por el hijo derecho del nodo actual.
3. El nodo buscado es el nodo donde termina el camino al haber recorrido todos los dígitos de la secuencia.

Por ejemplo, si buscamos el nodo número 25, los pasos anteriores serían las siguientes, que se corresponden con el recorrido gráfico que se puede ver a la derecha:

1. Comenzamos el recorrido en el nodo raíz del árbol binario y recorremos la secuencia de dígitos de la representación binaria de 25, que es 11001:
   a. Tenemos un 1, descendemos por el hijo derecho.
   b. Tenemos un 1, descendemos por el hijo derecho.
   c. Tenemos un 0, descendemos por el hijo izquierdo.
   d. Tenemos un 0, descendemos por el hijo izquierdo.
   e. Tenemos un 1, descendemos por el hijo derecho.
2. Hemos llegado al nodo buscado.



Para que esto funcione correctamente, la operación `set` deberá encargarse de añadir tantos nodos como sean necesarios para completar el recorrido desde la raíz al nodo que contendrá el elemento a insertar. De igual manera, la operación `delete` se encargará de borrar todos los nodos que ya no sean necesarios tras eliminar el nodo que contenía el elemento borrado.

2.1 Ejemplo de funcionamiento

A continuación vamos a dar un ejemplo del funcionamiento de la práctica. Junto a este enunciado se proporcionan algunos ficheros de test que pueden ser cargados desde la práctica. En nuestro ejemplo el programa carga el fichero de test `test01.txt` y lo ejecuta:

```
*Main> main
Nombre del fichero de test: test01.txt
Loading test file test01.txt
GET 1: Null
SET 1 A
GET 1: "A"
SET 10 B
ERROR: instrucción "COSA" no definida
GET 10: "B"
SET 10 NUEVO
DELETE 1
GET 1: Null
GET 10: "NUEVO"
DELETE 10
Nombre del fichero de test:
```

Cada fichero de test contiene una secuencia de instrucciones GET, SET y DELETE que se deberán ejecutar sobre un array disperso inicialmente vacío. El programa carga dicha secuencia y ejecuta las

operaciones en el orden indicado, mostrando el resultado de cada operación (si este existe). A continuación se puede ver la explicación de los resultados obtenidos por cada instrucción:

- `GET 1: Null` (porque inicialmente el array está vacío)
- `SET 1 A` (metemos "A" en la posición 1)
- `GET 1: "A"` (consultamos la posición 1 y obtenemos "A")
- `GET 1: Null` (porque inicialmente el array está vacío)
- `SET 10 B` (metemos "B" en la posición 10)
- `ERROR: instrucción "COSA" no definida` (ejemplo de error por instrucción errónea)
- `GET 10: "B"` (consultamos la posición 10 y obtenemos "B")
- `SET 10 NUEVO` (cambiamos el contenido de la posición 10 a "NUEVO")
- `DELETE 1` (eliminamos la posición 1)
- `GET 1: Null` (consultamos la posición 1 y obtenemos Null)
- `GET 10: "NUEVO"` (consultamos la posición 10 y obtenemos "NUEVO")
- `DELETE 10` (eliminamos la posición 10)

Cuando no se desea cargar más ficheros, sólo hay que dejar el nombre del fichero vacío y finalizará la ejecución del programa:

```
Nombre del fichero de test:
Game Over.
*Main>
```

2.3 Estructura de módulos de la práctica

La práctica está dividida en dos módulos, cada uno en un fichero independiente cuyo nombre coincide (incluyendo mayúsculas y minúsculas) con el nombre del módulo y cuya extensión es `.hs`:

1. Módulo `SparseArray`: dentro de este módulo se programarán las funciones indicadas para poder operar con los arrays dispersos. Incluye las definiciones de todos los tipos de datos utilizados en la práctica, además de una función que devuelve la representación binaria de un número en forma de lista de booleanos (siendo `False` un `0` y `True` un `1`).
2. Módulo `Main`: este módulo contiene la función principal y las encargadas de interactuar con el usuario y cargar los ficheros de test. Este módulo se proporciona ya totalmente programado y no deberá ser modificado.

Dado que un estudio más profundo de los módulos en **Haskell** está fuera del ámbito de la asignatura, sólo indicaremos que el módulo `Main` importa (además del módulo `SparseArray`) dos módulos adicionales para poder trabajar más cómodamente con la entrada/salida y poder acceder a ficheros.

2.4 Programa principal (módulo Main)

El módulo `Main` contiene el programa principal, ya programado por el equipo docente. Hay

funciones que utilizan mónadas y están escritas utilizando la "*notación do*", que es una notación para facilitar la escritura de concatenaciones de funciones monádicas. Sin entrar en detalle sobre el funcionamiento de las mónadas, vamos a explicar qué hace cada función:

- `loadFile`: carga el fichero de texto indicado con un test y devuelve su contenido (de tipo `String`).
- exec: esta función recibe una cadena de texto con un fichero de test y lo ejecuta. Para ello:
    1. Crea un array disperso de `String` vacío.
    2. Separa el fichero en líneas.
    3. Separa cada línea en tokens separados por espacios en blanco:
        3.1. El primer token es la instrucción a ejecutar (`set`, `get` o `delete`).
        3.2. El segundo token es el índice sobre el que ejecutar la instrucción.
        3.3. El tercer token es el elemento a insertar (sólo usado por `set`).
    4. Ejecuta cada instrucción sobre el array disperso modificando su estado y compone la salida mostrando la instrucción ejecutada y el resultado de la misma (si existe).
    5. Muestra un error si se lee una instrucción no definida, pero no detiene la ejecución del fichero en este caso.
- `loop`: esta función es el "*bucle principal*" del programa. Pregunta al usuario el nombre del fichero con el test a cargar, lo carga mediante una llamada a la función `loadFile` e imprime el resultado de la ejecución del test. Cuando el usuario introduce un nombre de fichero vacío, finaliza la ejecución del programa.
- `main`: es la función principal. Configura la entrada para que se pueda editar (si no, no tendría efecto la tecla de retroceso) y llama a la función `loop`.

Como ya se ha indicado, este módulo se entrega ya programado por el equipo docente. Para su correcto funcionamiento hay que programar las funciones necesarias en el módulo `SparseArray`.

2.5 Implementación (módulo SparseArray)

En este apartado vamos a mostrar las cuatro funciones que deberán ser programadas para trabajar con los arrays dispersos. En primer lugar, veamos los tipos de datos (ya programados):

```
data Value a = Null | Value a
  deriving (Eq,Read,Show)
data SparseArray a = Vacio |
                     Nodo (Value a) (SparseArray a) (SparseArray a)
  deriving (Eq,Read,Show)
```

- El tipo `Value a` representa un posible valor de cualquier tipo almacenado en el array disperso. Se le añade el valor `Null` para representar el contenido de los índices bajo los que no se ha almacenado ningún valor.
- El tipo `SparseArray a` representa un array disperso de cualquier tipo. Un array disperso será o bien el array disperso vacío (representado por el valor `Vacio`) o bien un nodo conteniendo un valor (de tipo `Value a`) y dos arrays dispersos del mismo tipo. Es decir, el array disperso se implementa mediante un árbol binario.

Para que el programa pueda realizar su trabajo es necesario programar las siguientes cuatro funciones:

1. **`newSparseArray :: Eq a => SparseArray a`**: que deberá devolver un array disperso vacío

2. **`set :: Eq a => SparseArray a -> Int -> a -> SparseArray a`**: que recibe un array disperso `sa`, un entero `idx` y un elemento `elem` y devuelve el array disperso resultado de modificar el contenido del array `sa` de manera que el elemento `elem` se encuentre indexado bajo el índice `idx`. Esta función deberá crear cuantos nodos intermedios sean necesarios en la estructura para asegurarse de que el nodo que debe contener el valor asociado al índice `idx` está en su lugar adecuado.
3. **`get :: Eq a => SparseArray a -> Int -> (Value a)`**: que recibe un array disperso `sa` y un entero `idx` y devuelve el valor de `sa` indexado bajo el índice `idx`. Si no hubiera ningún valor indexado bajo `idx`, se devolverá `Null`.
4. **`delete :: Eq a => SparseArray a -> Int -> SparseArray a`**: que recibe un array disperso `sa` y un entero `idx` y devuelve el array disperso resultado de eliminar de `sa` el valor indexado bajo el índice `idx`. Si en `sa` no hubiera ningún valor indexado bajo `idx`, el resultado de la función sería el propio array disperso de entrada `sa`. Esta función deberá eliminar los nodos vacíos que resulten de la eliminación del nodo asociado al índice `idx`.