5. Un *Diccionario* es una estructura de datos muy utilizada en programación. Consiste de una colección de pares (Clave,Valor), a la cual le puedo realizar las operaciones:

   - Crear un diccionario vacío.
   - Agregar el par consistente de la clave k y el valor v. En caso que la clave ya se encuentre en el diccionario, se reemplaza el valor asociado por v.
   - Chequear si un diccionario es vacío.
   - Chequear si una clave se encuentra en el diccionario.
   - Buscar el valor asociado a una clave k. Solo se puede aplicar si la misma se encuentra.
   - Una operación que dada una clave k, elimina el par consistente de k y el valor asociado. Solo se puede aplicar si la clave se encuentra en el diccionario.
   - Una operación que devuelve un conjunto con todas las claves contenidas en un diccionario.

   (a) Especificá el TAD diccionario indicando constructores y operaciones.

   spec Dict **of** (K,V) **where**

   donde K y V pueden ser cualquier tipo, asegurando que K tenga definida una función que chequea igualdad.

```
spec Dict of (K,V) where

constructors
    fun empty_dict() ret d: Dict of (K,V)
    {- crea un diccionario vacío -}

    proc add_to_dict(in/out d: Dict of (K,V), in k: K, in v: V)
    {- agrega el par (k,v) a d. Si la clave k ya se encuentra en d, se reemplaza el valor por v -}

destroy
    proc destroy_dict(in/out d: Dict of (K,V))
    {- devuelve true si y solo si el diccionario es vacío -}

operations
    fun is_empty_dict(d: Dict of (K,V)) ret b: bool
    {- devuelve true si y solo si el diccionario es vacío -}

    fun key_in_dict(d: Dict of (K,V), k: K) ret b: bool
    {- devuelve true si y solo si la clave k está en el diccionario d -}

    {- PRE: key_in_dict(d,k) -}
    fun value_of_k(d: Dict of (K,V), k: K) ret v: V
    {- busca el valor asociado v de una clave k en el diccionario d -}

    {- PRE: key_in_dict(d,k) -}
    proc delete_key_and_value(in/out d: Dict of (K,V), in k: K)
    {- elimina el par consistente de k y su valor asociado v -}

    fun set_of_keys(d: Dict of (K,V)) ret s: Set of K
    {- devuelve un conjunto con todas las claves contenidas en d -}

end spec
```

(b) Implementá el TAD diccionario utilizando la siguiente representación:

implement Dict **of** (K,V) **where**

type Node **of** (K,V) = **tuple**
left: **pointer to** (Node **of** (K,V))
key: K
value: V
right: **pointer to** (Node **of** (K,V))
end tuple

type Dict **of** (K,V)= **pointer to** (Node **of** (K,V))

Como invariante de representación debemos asegurar que el árbol representado por la estructura sea binario de búsqueda de manera que la operación de buscar un valor tenga orden logarítmico. Es decir, dado un nodo n, toda clave ubicada en el nodo de la derecha n.right, debe ser mayor o igual a n.key. Y toda clave ubicada en el nodo de la izquierda n.left, debe ser menor a n.key. Debes tener especial cuidado en la operación que agrega pares al diccionario.

```
implement Dict of (K,V) where

type Node of (K,V) = tuple
                       left: pointer to (Node of (K,V))
                       key: K
                       value: V
                       right: pointer to (Node of (K,V))
                     end tuple

type Dict of (K,V) = pointer to (Node of (K,V))

constructors
    fun empty_dict() ret d: Dict of (K,V)
        d := null
    end fun

    proc add_to_dict(in/out d: Dict of (K,V), in k: K, in v: V)
        if is_empty_dict(d) then
            alloc(d)
            d→left := null
            d→key := k
            d→value := v
            d→right := null
        else
            if k = d→key then
                d→value := v
            else if k < d→key then
                add_to_dict(d→left,k,v)
            else if k > d→key then
                add_to_dict(d→right,k,v)
            fi
        fi
    end proc

destroy
    proc destroy_dict(in/out d: Dict of (K,V))
        if not is_empty_dict(d) then
            destroy_dict(d→left)
            destroy_dict(d→right)
```

```
            free(d)
        fi
    end proc

operations
    fun is_empty_dict(d: Dict of (K,V)) ret b: bool
        b := (d = null)
    end fun

    fun key_in_dict(d: Dict of (K,V), k: K) ret b: bool
        if is_empty_dict(d) then
            b := false
        else
            if k = d→key then
                b := true
            else if k < d→key then
                key_in_dict(d→left,k)
            else if k > d→key then
                key_in_dict(d→right,k)
            fi
        fi
    end fun

    {- PRE: key_in_dict(d,k) -}
    fun value_of_k(d: Dict of (K,V), k: K) ret v: V
        if k = d→key then
            v := d→value
        else if k < d→key then
            value_of_k(d→left,k)
        else if k > d→key then
            value_of_k(d→right,k)
        fi
    end fun

    {- PRE: key_in_dict(d,k) -}
    proc delete_key_and_value(in/out d: Dict of (K,V), in k: K)
        var temp: Dict of (K,V)

        if k = d→key then
            temp := d→right
            d := d→left
            union_branches(d,temp)
            destroy_dict(temp)
        else if k < d→key then
            delete_key_and_value(d→left,k)
        else if k > d→key then
            delete_key_and_value(d→right,k)
        fi
    end proc

    proc union_branches(in/out d1: Dict of (K,V), in d2: Dict of (K,V))
        if not is_empty_dict(d1) then
            add_to_dict(d1,d2→key,d2→value)
            union_branches(d1,d2→left)
            union_branches(d1,d2→right)
        fi
    end proc
```

```
    fun set_of_keys(d: Dict of (K,V)) ret s: Set of K
        s := empty_set()

        if not is_empty_dict(d) then
            add(s,d→key)
            set_of_keys(d→left)
            set_of_keys(d→right)
        fi
    end fun

end implement
```