Nombre:	LU:		Cant. hojas:

Estructuras de Datos - Segundo Parcial

Licenciatura en Ciencias de la Computación – Ingeniería en Computación – Ingeniería en Sistemas de Software
Universidad Nacional del Sur – 23/10/2025

Observaciones generales:

- **REALICE LOS EJERCICIOS EN HOJAS SEPARADAS**.
- Lea todo el ejercicio antes de comenzar a desarrollarlo.
- Numere y ponga su nombre a todas las hojas. Indique cuántas hojas entrega.
- Recuerde que se evalúan correctitud, eficiencia y legibilidad de sus soluciones.

Importante: para resolver este examen utilice las interfaces presentadas en clase. Al final del examen se encuentra un recordatorio de los métodos que cada una provee.

Ejercicio 1: a) Suponga que cuenta con la implementación de la clase DiccionarioHash que implementa un diccionario utilizando la estrategia de hash abierto. Agregue un método a esta clase con la siguiente signatura: **`public int todas(K key) throws InvalidKeyException`**. Este método deberá retornar un número entero con la cantidad de entradas con clave equivalente a **key**. Si utiliza otros métodos del TDADiccionario deberá implementarlos. Si utiliza métodos auxiliares deberá implementarlos.
b)Indique el orden del tiempo de ejecución de su solución. Justifique adecuadamente.

```java
/**
     * Retorna un iterable con todas las entradas que tienen la clave key.
     *
     * @param key La clave a buscar
     * @return Iterable con todas las entradas que tienen esa clave
     * @throws InvalidKeyException si la clave es inválida (null)
     */
    public int todas(K key) throws InvalidKeyException {
        // Validar que la clave no sea nula
        if (key == null) {
            throw new InvalidKeyException("La clave no puede ser nula");
        }

        int cantidad = 0;

        // Calcular el índice del bucket donde estaría la clave
        int indice = hashThisKey(key);

        // Obtener la lista de ese bucket
        PositionList<Entry<K, V>> lista = bucket[indice];

        // Recorrer la lista buscando todas las entradas con la clave key
        for (Entry<K, V> entrada : lista) {
            if (entrada.getKey().equals(key)) {
                // Agregar la entrada al resultado
                cantidad ++;
            }
        }
```

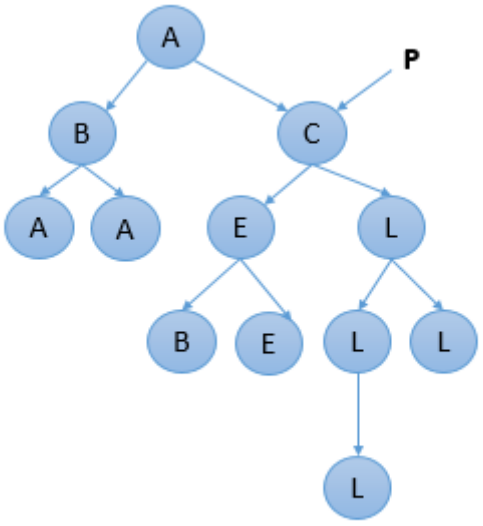
```
        return cantidad;
    }

     /**
      * Calcula funci�n hash a partir de la clave k dada.
      * @param k clave.
      * @return posici�n del "bucket" donde se ubicar� la entrada con clave k.
      * @throws InvalidKeyException
      */
     private int hashThisKey(K k) throws InvalidKeyException{
           if (k==null){
                 throw new InvalidKeyException ("Error en Mapeo: Clave nula");
           }
           return Math.abs(k.hashCode() % N); //por si acaso pasan un negativo.
     }
```

b- O(1). En el caso promedio el orden es O(1)(Constante)
Este es el objetivo de una tabla hash. Si la función de hash distribuye bien las claves y el mapa no está demasiado "lleno" (es decir, el factor de carga es bajo), cada bucket tendrá muy pocos elementos (idealmente 0 o 1).

Ejercicio 2: Escriba un método tal que dado un árbol binario de caracteres y una posición p elimine del subarbol p todas las hojas y retorne un mapeo con cada caracter eliminado y la cantidad de veces que estaba en el subárbol.
Signature:

```
public Mapeo<Character, Integer> eliminarHojas(BinaryTree<Character>
arbol,Position<Character> p) throws InvalidPositionException
```

Asuma que cuenta con la implementación de los TDAs ArbolBinario y Mapeo completas. Si utiliza métodos auxiliares deberá implementarlos. Resuelva el ejercicio realizando un recorrido en **posorden**. Ejemplo: para el árbol que se muestra en la figura el mapeo resultante deberá ser={(B,1),(E,1),(L,2)}.



```
  public Mapeo<Character, Integer> eliminarHojas(BinaryTree<Character> arbol,
Position<Character> p) throws InvalidPositionException {
   Mapeo<Character, Integer> resultado = new MapeoConHashAbierto<>(); // O la
implementación disponible
   eliminarHojasRec(arbol, p, resultado);
   return resultado;
}
/**
* Método auxiliar recursivo que realiza el recorrido en posorden
* para eliminar las hojas y contarlas en el mapeo.
*/
private void eliminarHojasRec(BinaryTree<Character> arbol, Position<Character> p,
Mapeo<Character, Integer> mapeo) throws InvalidPositionException {

//guardo si es hoja antes del llamado recursivo para evitar eliminarlo si no lo era.
bool eliminarNodo = (!arbol.hasLeft(p) && !arbol.hasRight(p));

if (p == null) {
        return;
   }

   // Posorden: primero visito los hijos (izquierdo y derecho), luego el nodo actual

   // Procesar hijo izquierdo si existe
   if (arbol.hasLeft(p)) {
       Position<Character> hijoIzq = arbol.left(p);
       eliminarHojasRec(arbol, hijoIzq, mapeo);
   }

   // Procesar hijo derecho si existe
   if (arbol.hasRight(p)) {
       Position<Character> hijoDer = arbol.right(p);
       eliminarHojasRec(arbol, hijoDer, mapeo);
   }

   // Ahora proceso el nodo actual (característica del posorden)
   // Verifico si es una hoja (después de procesar sus hijos)
   if (eliminarNodo){
       Character caracter = p.element();

       // Actualizar el mapeo con el conteo
       Integer conteo = mapeo.get(caracter);
      if (conteo != null){
           mapeo.put(caracter, conteo + 1);
       } else {
              mapeo.put(caracter, 1);
       }

        arbol.remove(p);
       }
   }
}
```

Ejercicio 3: Suponga que cuenta con la implementación de la clase árbol que implementa un árbol general con nodos enlazados tal como se vio en clase (enlace al padre y lista de hijos). Agregue a esta clase un método con la siguiente signatura: `public boolean eliminarUltimo(Position<E> p) throws InvalidPositionException`. Este método deberá eliminar la posición p del árbol si y solo si es el último hijo de su padre (si p tiene hijos, ponerlos ordenadamente en el lugar de su padre tal como lo hace el remove). Retorna verdadero si se pudo eliminar y falso en caso contrario. Si utiliza otros métodos del TDAArbol deberá implementarlos. Si utiliza métodos auxiliares deberá implementarlos.

```
public class Arbol<E> implements Tree<E> {
   protected TNodo<E> root;
   protected int size;

   /**
    * Elimina la posición p del árbol si y solo si es el último hijo de su padre.
    * @param p La posición a eliminar
    * @return true si se pudo eliminar, false en caso contrario
    * @throws InvalidPositionException si la posición es inválida
    */
   public boolean eliminarUltimo(Position<E> p) throws InvalidPositionException {
       TNodo<E> nodo = checkPosition(p);


      // No se puede eliminar la raíz con este método
       if (nodo == root) {
           return false;
       }

       // Obtener el padre del nodo
       TNodo<E> padre = nodo.getPadre();

       // Verificar que el padre no sea nulo (por seguridad)
       if (padre == null) {
           return false;
       }

       // Obtener la lista de hijos del padre
       PositionList<TNodo<E>> hijos = padre.getHijos();

       // Verificar que el padre tenga hijos
       if (hijos.isEmpty()) {
           return false;
       }

       // Obten60er la última posición de la lista
       Position<TNodo<E>> ultimaPosicion = hijos.last();

       // Verificar si el nodo es el último hijo
       // Nota: last() retorna la posición del último elemento antes del trailer
       if (ultimaPosicion.element() == nodo) {
           // El nodo es el último hijo, proceder a eliminarlo

           // Verificar que el nodo no tenga hijos (opcional, según la especificación)
           // Si tiene hijos, ubicar a los hijos en el lugar del padre ordenados.


            hN = nodo.getHijos();
            while(!hN.isEmpty())
                  {
                        Position<TNodo<E>> hijoN=hN.first();
                        padre.addBefore(ultimaPosicion,hijoN.element());
                        hijoN.element().setPadre(padre);
```

```
                    hN.remove(hijoN);
                }

            // Eliminar el nodo de la lista de hijos del padre
            hijos.re60move(ultimaPosicion);

            // Desconectar el nodo (buena práctica)
            nodo.setPadre(null);

            // Decrementar el tamaño del árbol
            size--;

                return true;
            }


        // El nodo no es el último hijo
        return false;
    }

    /**
     * Método auxiliar para validar y convertir una Position a TNodo
     * @param p La posición a validar
     * @return El nodo correspondiente
     * @throws InvalidPositionException si la posición es inválida
     */
    private TNodo<E> checkPosition(Position<E> p) throws InvalidPositionException {
        if (p == null) {
            throw new InvalidPositionException("Posición nula");
        }

        try {
            TNodo<E> nodo = (TNodo<E>) p;

            // Verificar que el nodo pertenezca a este árbol
            // (verificación adicional de seguridad)
            if (nodo.getPadre() == null && nodo != root) {
                throw new InvalidPositionException("El nodo no pertenece a este árbol");
            }

            return nodo;
        } catch (ClassCastException e) {
            throw new InvalidPositionException("Tipo de posición inválido");
        }
    }
}
```

PositionList<E>:	**Tree<E>**	**BinaryTree<E>**	**Map<K,V>:**	**Dictionary<K,V>**
size( )	size()	left(p)	size()	size()
isEmpty()	isEmpty()	right(p)	isEmpty()	isEmpty()
first()	iterator()	hasLeft(p)	get(k)	find(K key)
last()	positions()	hasRigth(p)	put(k, v)	findAll(K key)
next(p)	replace(v, e)	createRoot(r)	remove(k)	insert(K key, V value)
prev(p)	root()	addLeft(r,p)	keys()	remove(Entry<K,V> e)
addFirst(e)	parent(v)	addRigth(r,p)	values()	entries()
addLast(e)	children(v)	remove(p)	entries()	
addAfter(p, e)	isInternal(v)	attach(p,t1,t2)		
addBefore(p, e)	isExternal(v)			
remove(p)	isRoot(v)			
set (p, e)	createRoot(e)			
iterator()	addFirstChild(p,e)			
positions()	addLastChild(p,e)			
	addBefore(p,rb,e)			
	addAfter (p,lb,e)			

	removeExternalNode (p) removeInternalNode (p) removeNode (p)			