

Nombre y Apellido: Brazza Santiago N° Legajo: 57435

Parcial Bis de Estructuras de Datos y Algoritmos

Segundo Cuatrimestre de 2017

Ejercicio 1	Ejercicio 2	Ejercicio 3	Ejercicio 4	Ejercicio 5	Nota
B	R	B	B-	R	7,5

Condición Mínima de Aprobación: Tener por lo menos tres ejercicios con B-

Consideraciones a tener en cuenta. MUY IMPORTANTE

- El ejercicio que no respete estrictamente el enunciado será anulado.
- Puede entregar el examen escrito en lápiz
- Realizar cada ejercicio en hojas separadas
- Se tendrán en cuenta la eficiencia y el estilo de programación.
- Los teléfonos celulares deben estar apagados.

Ejercicio 1

Se cuenta con la siguiente interfaz que representa una lista lineal simplemente encadenada:

```
public interface LinkedList<T> {  
  
    /** Agrega un elemento al final de la lista. */  
    public void add(T value);  
  
    /** Elimina un elemento de la lista, según el índice que ocupe.  
     * Si el índice es inválido no hace nada. */  
    public void remove(int index);  
  
    /** Imprime por consola todos los elementos de la lista. */  
    public void print();  
  
    /** deshace la última operación de remove realizada sobre la lista.  
     * El elemento que había sido eliminado vuelve a aparecer, en la posición  
     * que ocupaba originalmente. */  
    public void undoRemove();  
  
}
```

A continuación se muestra un ejemplo de uso:

```
public static void main(String[] args) {  
    LinkedList<String> list = new LinkedListImpl<String>();  
  
    list.add("A");  
    list.add("B");  
    list.add("C");  
    list.add("D");  
    list.print(); // A B C D  
    list.remove(3);  
    list.remove(1);  
    list.print(); // A C  
    list.remove(1);  
    list.undoRemove();  
    list.undoRemove();  
    list.print(); // A B C  
    list.add("E");  
    list.print(); // A B C E  
    list.undoRemove();  
    list.print(); // A B C D E  
}
```

Realizar una implementación de *LinkedList*. Los métodos *add* y *undoRemove* deben resolverse con complejidad temporal $O(1)$. **No utilizar clases de la API de Java.**

Ejercicio 2

En el juego *Find The Gold*, un minero tiene que ir moviéndose por una grilla (solo para arriba, izquierda, derecha o abajo) para llegar al oro. En cada celda de la grilla puede estar el oro (marcado con 'G') o puede haber rocas ('R') que imposibilitan el paso del minero. El minero tiene una cantidad máxima de pasos que puede dar antes de perder.

Dado:


- Un tablero de $N \times M$ donde existe una celda con oro y múltiples rocas;
- La posición inicial del minero;
- La cantidad máxima de pasos que puede dar el minero

Retornar *True* si el minero puede llegar hasta el oro con esa cantidad de pasos, o sino, *False*.

```
public static boolean canFindGold(char[][] board, Point initial, int maxSteps);
```


Por ejemplo:

Para el siguiente tablero, con un $\text{maxSteps} \geq 5$ debe retornar *True*. En caso contrario, *False*.

			
R	R		
R			
	G	R	

Ejercicio 3

Dada la siguiente implementación de árbol.

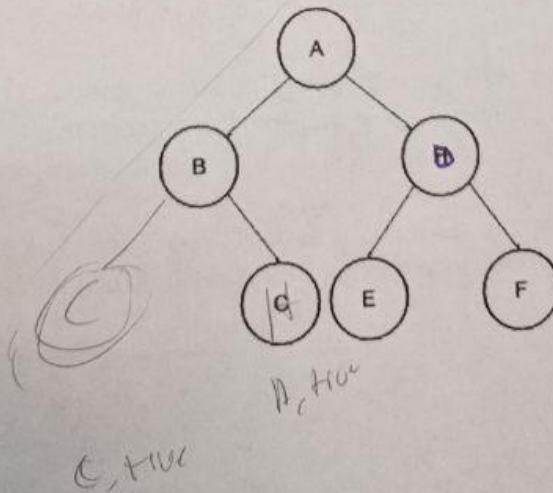
```
public class BinaryTree<V> {  
    V value;  
    BinaryTree left;  
    BinaryTree right;  
}
```

Implementar un método que dado un árbol y un comparador, retorne si el recorrido *preorder* del árbol se encuentra ordenado.

```
public static boolean preOrderSorted(BinaryTree tree, Comparator<V> cmp);
```

Por ejemplo:

Para el siguiente árbol, debe retornar *True*.

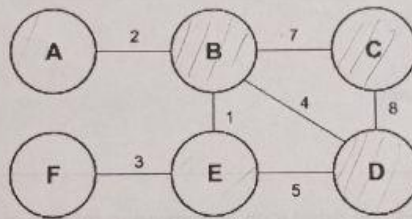


Ejercicio 4

Dado un grafo con pesos **positivos** en las aristas, implementar un algoritmo que, dado un valor N , determine si existe un camino entre cualquier par de nodos de longitud menor o igual a N .

Por ejemplo:

Para el siguiente grafo, si se lo llama con $N=8$, debería retornar *False* pues no existe camino entre A y C con longitud < 8 . Si se lo llama con $N=10$, debe retornar *True*.



Ejercicio 5

Se tiene que elegir un equipo de M personas de una empresa que cuenta con N personas ($N > M$) para desarrollar un nuevo proyecto. Se busca que el equipo tenga la mayor *performance* posible y para eso se cuenta con un método *getPerformance* que recibe las personas del equipo y, retorna un número entre 0 y 1 con la *performance* esperada de dicho equipo.

Como N y M son valores altos, no es posible generar todos los posibles equipos y calcular su *performance*. Es por eso que se pide implementar una estrategia *Hill Climbing* para encontrar el mejor equipo posible, sabiendo que se cuenta con un 48 horas de una CPU exclusiva antes de que arranque el proyecto.

La elección de las estructuras de datos y de la formalización de los parámetros que manejan los métodos (tanto *getPerformance* como los otros que hay que implementar) es libre.