Final Regular Ejercicios

1. Representación de Grafos

a. Dibuje una representación con estructura dinámica para los siguientes grafos:

Para dibujar una representación con estructura dinámica de un grafo, usualmente utilizamos listas de adyacencia. Cada nodo en el grafo tiene una lista que contiene sus nodos adyacentes.

1. Grafo no dirigido:
   - Nodos: A, B, C, D
   - Aristas: (A-B), (A-C), (B-D), (C-D)

   Representación con listas de adyacencia:

   ```
   A: [B, C]
   B: [A, D]
   C: [A, D]
   D: [B, C]
   ```

2. Grafo dirigido:
   - Nodos: A, B, C, D
   - Aristas: A→B, B→C, C→D, D→A

   Representación con listas de adyacencia:

   ```
   A: [B]
   B: [C]
   C: [D]
   D: [A]
   ```

b. Representación estática y dinámica en Java

Representación estática (matriz de adyacencia):

```
int[][] grafo = {
    {0, 1, 1, 0}, // A
    {1, 0, 0, 1}, // B
    {1, 0, 0, 1}, // C
    {0, 1, 1, 0}  // D
};
```

Representación dinámica (listas de adyacencia):

```
import java.util.ArrayList;
import java.util.List;

class Grafo {
    private List<List<Integer>> adjList;

    public Grafo(int numVertices) {
        adjList = new ArrayList<>();
        for (int i = 0; i < numVertices; i++) {
            adjList.add(new ArrayList<>());
        }
    }

    public void agregarArista(int origen, int destino) {
        adjList.get(origen).add(destino);
        adjList.get(destino).add(origen); // Para grafo no
dirigido
    }

    public List<Integer> obtenerAdyacentes(int vertice) {
        return adjList.get(vertice);
    }
}
```

c. Método que devuelve los nodos del grafo

```
import java.util.LinkedList;

public LinkedList<Integer> obtenerNodos(Grafo grafo) {
    LinkedList<Integer> nodos = new LinkedList<>();
    for (int i = 0; i < grafo.adjList.size(); i++) {
        nodos.add(i);
    }
    return nodos;
}
```

Complejidad temporal: O(V), donde V es el número de vértices en el grafo.

d. Método que devuelve los nodos adyacentes a un nodo dado

```
public LinkedList<Integer> obtenerAdyacentes(Grafo grafo, i
nt nodo) {
    LinkedList<Integer> adyacentes = new LinkedList<>(graf
o.obtenerAdyacentes(nodo));
    return adyacentes;
}
```

Complejidad temporal: O(A), donde A es el número de adyacentes al nodo dado.

2. Implementación de la clase ABB (Árbol Binario de Búsqueda)

```
public interface ABBTDA{
    void inicializarArbol();
    int raiz();
    boolean arbolVacio(); // arbol debe estar inicializado
    ABBTDA hijoDer();
    ABBBTDA hijoIzq();
    void agregarElem(int x);
    void eliminarElem(int x); // arbol no vacio e inicializ
ado
}
```

```
public class ABB implements ABBTDA {
    private class NodoABB{
        int info;
        ABBTDA hI;
        ABBTDA hD;
    }

    private NodoABB a;

    public void inicializarArbol(){
        a = null;
    }

    public int raiz(){
        return a.info();
    }

    public boolean arbolVacio(){
        return (a == null);
    }

    public ABBTDA hijoDer(){
        return a.hD;
    }

    public ABBTDA hijoIzq(){
        return a.hI;
    }

    private int mayor(ABBTDA a){
        if(a.hijoDer().arbolVacio()){
            return a.raiz();
        }else{
            return mayor(a.hijoDer());
        }
    }

    private int menor(ABBTDA a){
```

```
        if(a.hijoIzq().arbolVacio()){
            return a.raiz();
        }else{
            return menor(a.hijoIzq());
        }
    }

    public void agregarElem(int x){
        if(a == null) { // arbol sin empezar
            a = new NodoABB();
            a.info = x;
            a.hI = new ABB();
            a.hI.inicializarAbol();
            a.hD = new ABB();
            a.hD.incializarArbol();
        }else (a.info >x){
            a.hI.agregarElem(x);
        }
    }

    public void eliminarElem(int x){
        if(a != null){
            if(a.info == x && a.hI.arbolVacio() && a.hD.arb
olVacio() ){
                a = null;
            }else if (a.info == x && !a.hI.arbolVacio()){
                int mayor = mayor(a.hI);
                a.info = mayor;
                a.hI.eliminarElem(mayor);
            } else if (a.info == x && !a.hD.arbolVacio() ){
                int menor = menor(a.hD);
                a.info = menor;
                a.hD.eliminarElem(menor);
            } else if (a.info < x){
                a.hD.eliminarElem(x);
            } else {
                a.hI.eliminarElemn(x);
            }
```

```
        }

    }

}
```

3. Recorridos pre-order y post-order de un Árbol Binario

```
class ArbolBinario {
    Nodo raiz;

    void preOrder(Nodo nodo) {
        if (nodo == null)
            return;
        System.out.print(nodo.valor + " ");
        preOrder(nodo.izquierdo);
        preOrder(nodo.derecho);
    }

    void postOrder(Nodo nodo) {
        if (nodo == null)
            return;
        postOrder(nodo.izquierdo);
        postOrder(nodo.derecho);
        System.out.print(nodo.valor + " ");
    }
}
```

Prueba de recorridos para un árbol:

```
      1
     / \\
    2   3
   / \\
  4   5
```

Salida pre-order: `1 2 4 5 3`
Salida post-order:
`4 5 2 3 1`

> Un Árbol Binario de Búsqueda (ABB) y un Árbol Binario son estructuras similares, pero el ABB tiene la propiedad adicional de que para cada nodo, todos los elementos en el subárbol izquierdo son menores y todos los elementos en el subárbol derecho son mayores.

Ejercicio 4: Método que devuelve las hojas de un árbol binario

```
import java.util.LinkedList;

public class ABB implements ABBTDA {
    private class NodoABB {
        int info;
        ABBTDA hI;
        ABBTDA hD;
    }

    private NodoABB a;

    public void inicializarArbol() {
        a = null;
    }

    public int raiz() {
        return a.info;
    }

    public boolean arbolVacio() {
        return (a == null);
    }

    public ABBTDA hijoDer() {
```

```
        return a.hD;
    }

    public ABBTDA hijoIzq() {
        return a.hI;
    }

    private int mayor(ABBTDA a) {
        if (a.hijoDer().arbolVacio()) {
            return a.raiz();
        } else {
            return mayor(a.hijoDer());
        }
    }

    private int menor(ABBTDA a) {
        if (a.hijoIzq().arbolVacio()) {
            return a.raiz();
        } else {
            return menor(a.hijoIzq());
        }
    }

    public void agregarElem(int x) {
        if (a == null) {
            a = new NodoABB();
            a.info = x;
            a.hI = new ABB();
            a.hI.inicializarArbol();
            a.hD = new ABB();
            a.hD.inicializarArbol();
        } else if (a.info > x) {
            a.hI.agregarElem(x);
        } else {
            a.hD.agregarElem(x);
        }
    }
```

```
    public void eliminarElem(int x) {
        if (a != null) {
            if (a.info == x && a.hI.arbolVacio() && a.hD.ar
bolVacio()) {
                a = null;
            } else if (a.info == x && !a.hI.arbolVacio()) {
                int mayor = mayor(a.hI);
                a.info = mayor;
                a.hI.eliminarElem(mayor);
            } else if (a.info == x && !a.hD.arbolVacio()) {
                int menor = menor(a.hD);
                a.info = menor;
                a.hD.eliminarElem(menor);
            } else if (a.info < x) {
                a.hD.eliminarElem(x);
            } else {
                a.hI.eliminarElem(x);
            }
        }
    }

    // Método para obtener las hojas del árbol
    public LinkedList<Integer> obtenerHojas() {
        LinkedList<Integer> hojas = new LinkedList<>();
        obtenerHojasRec(a, hojas);
        return hojas;
    }

    private void obtenerHojasRec(NodoABB nodo, LinkedList<I
nteger> hojas) {
        if (nodo != null) {
            if (nodo.hI.arbolVacio() && nodo.hD.arbolVacio
()) {
                hojas.add(nodo.info);
            }
            if (!nodo.hI.arbolVacio()) {
                obtenerHojasRec((NodoABB) nodo.hI, hojas);
            }
```

```
            if (!nodo.hD.arbolVacio()) {
                obtenerHojasRec((NodoABB) nodo.hD, hojas);
            }
        }
    }
}
```

Complejidad temporal: O(n), donde n es el número de nodos en el árbol.

Ejercicio 5: Método que devuelve las alturas de cada uno de los nodos internos

```
import java.util.LinkedList;

public class ABB implements ABBTDA {
    private class NodoABB {
        int info;
        ABBTDA hI;
        ABBTDA hD;
    }

    private NodoABB a;

    public void inicializarArbol() {
        a = null;
    }

    public int raiz() {
        return a.info;
    }

    public boolean arbolVacio() {
        return (a == null);
    }

    public ABBTDA hijoDer() {
        return a.hD;
    }
```

```
    public ABBTDA hijoIzq() {
        return a.hI;
    }

    private int mayor(ABBTDA a) {
        if (a.hijoDer().arbolVacio()) {
            return a.raiz();
        } else {
            return mayor(a.hijoDer());
        }
    }

    private int menor(ABBTDA a) {
        if (a.hijoIzq().arbolVacio()) {
            return a.raiz();
        } else {
            return menor(a.hijoIzq());
        }
    }

    public void agregarElem(int x) {
        if (a == null) {
            a = new NodoABB();
            a.info = x;
            a.hI = new ABB();
            a.hI.inicializarArbol();
            a.hD = new ABB();
            a.hD.inicializarArbol();
        } else if (a.info > x) {
            a.hI.agregarElem(x);
        } else {
            a.hD.agregarElem(x);
        }
    }

    public void eliminarElem(int x) {
        if (a != null) {
            if (a.info == x && a.hI.arbolVacio() && a.hD.ar
```

```
bolVacio()) {
                a = null;
            } else if (a.info == x && !a.hI.arbolVacio()) {
                int mayor = mayor(a.hI);
                a.info = mayor;
                a.hI.eliminarElem(mayor);
            } else if (a.info == x && !a.hD.arbolVacio()) {
                int menor = menor(a.hD);
                a.info = menor;
                a.hD.eliminarElem(menor);
            } else if (a.info < x) {
                a.hD.eliminarElem(x);
            } else {
                a.hI.eliminarElem(x);
            }
        }
    }



}
```