Recorrido por niveles y uso de helpers en estructuras de datos

1. Cuándo usar helpers (funciones auxiliares)

Los helpers o funciones auxiliares son métodos privados que complementan a métodos públicos. En el código proporcionado podemos observar varios casos donde resultan útiles:

Casos de uso para helpers:

- **Mantener una interfaz pública limpia**: Los métodos públicos exponen una interfaz simple mientras que los helpers manejan la complejidad interna.
- **Recursividad con parámetros adicionales**: Como se observa en los métodos de recorrido, donde se necesita una lista para acumular resultados.
- **Encapsular lógica recursiva**: Separar la inicialización de la ejecución recursiva.
- **Preservar invariantes**: Los métodos públicos pueden realizar comprobaciones (como `isEmpty()`) antes de llamar al helper.

Ejemplo del código:

```
// Método público con interfaz simple
public List<Integer> numerosImparesMayoresQuePreOrden(Integer n)
        {
    List<Integer> lista = new ArrayList();
    if(!this.isEmpty())
        numerosImparesMayoresQuePreOrden(n,lista);  // Helper
        hace el trabajo recursivo
    return lista;
}

// Helper con parámetros adicionales (la lista para acumular
        resultados)
private void numerosImparesMayoresQuePreOrden(Integer n,
        List<Integer> lista) {
    if((Integer) this.getData() > n & (Integer) this.getData() %
        2 != 0)
        lista.add((Integer) this.getData());

    List<GeneralTree<T>> hijos = this.getChildren();
    for(GeneralTree<T> hijo: hijos)
```

```
        hijo.numerosImparesMayoresQuePreOrden(n,lista);
}
```

Ejemplo del método altura():

```
// Método público con verificación inicial
public int altura() {
    if(this.isEmpty())
        return -1;
    return alturaHelper();  // Delega al helper la recursividad
}

// Helper recursivo que calcula la altura real
private int alturaHelper() {
    if(this.isLeaf())
        return 0;
    else {
        int altAct = 0;
        List<GeneralTree<T>> hijos = this.getChildren();
        for(GeneralTree<T> hijo: hijos)
            altAct = Math.max(altAct, hijo.alturaHelper());
        return altAct + 1;  // Suma 1 para incluir este nivel
    }
}
```

2. Cuándo usar for(int i=0; i < cola.size(); i++) para recorrer niveles

Este patrón es fundamental para procesar árboles por niveles cuando necesitamos mantener un control preciso sobre el procesamiento de cada nivel completo antes de pasar al siguiente.

Casos de uso:

- **Cuando necesitamos identificar el nivel actual**: Como en el método `nivel(T dato)` donde contamos en qué nivel se encuentra un elemento.
- **Cuando queremos procesar todos los nodos de un nivel antes de continuar**: Esto garantiza que los niveles se procesen completamente antes de pasar al siguiente.
- **Para calcular propiedades específicas de cada nivel**: Como en el método `ancho()` que necesita conocer el tamaño de cada nivel.

Ejemplo del código - método nivel(T dato):

```
public int nivel(T dato) {
    if(!this.isEmpty()) {
        Queue<GeneralTree<T>> cola = new Queue<>();
        cola.enqueue(this);
        GeneralTree<T> aux;
        int level = 0;

        while(!cola.isEmpty()) {
            // Importante: capturamos el tamaño actual de la cola
        (nodos en este nivel)
            int currentLevelSize = cola.size();
            for(int i=0; i < currentLevelSize; i++) {
                aux = cola.dequeue();
                if(!aux.isEmpty() && aux.getData() == dato)
                    return level;  // Encontramos el dato,
        devolvemos el nivel actual
                for(GeneralTree<T> hijo: aux.getChildren())
                    cola.enqueue(hijo);
            }
            level++;  // Incrementamos el nivel después de
        procesar todo el nivel actual
        }
    }
    return -1;  // El dato no está en el árbol
}
```

Ejemplo del código - método ancho():

```
public int ancho() {
    if(!this.isEmpty()) {
        int anchoMax = -1;
        Queue<GeneralTree<T>> cola = new Queue<>();
        GeneralTree<T> aux;
        cola.enqueue(this);

        while(!cola.isEmpty()) {
            // Registramos el tamaño del nivel actual
            int currentLevelSize = cola.size();
            anchoMax = Math.max(anchoMax, currentLevelSize);

            // Procesamos exactamente los nodos del nivel actual
            for(int i=0; i < currentLevelSize; i++) {
                aux = cola.dequeue();
```

```
                for(GeneralTree<T> hijo: aux.getChildren())
                    cola.enqueue(hijo);
            }
        }
        return anchoMax;
    }
    return -1;
}
```

3. Recorrido de niveles con cola (BFS)

El recorrido por niveles o Breadth-First Search (BFS) utiliza una cola para visitar los nodos en orden de distancia desde la raíz, nivel por nivel.

Implementación básica para árboles generales:

```
public List<Integer> numerosImparesMayoresQuePorNiveles(Integer
        n) {
    List<Integer> lista = new ArrayList();
    Queue<GeneralTree<T>> cola = new Queue<>();
    cola.enqueue(this);
    GeneralTree<T> t;

    while(!cola.isEmpty()) {
        t = cola.dequeue();
        // Procesamos el nodo actual
        if(!t.isEmpty() & (Integer) t.getData() > n & (Integer)
        t.getData() % 2 != 0)
            lista.add((Integer) t.getData());
        // Añadimos todos los hijos a la cola para procesarlos
        después
        for(GeneralTree<T> hijo: t.getChildren())
            cola.enqueue(hijo);
    }

    return lista;
}
```

Variante para procesar nivel por nivel:

```
public void recorrerPorNiveles() {
    if(this.isEmpty()) return;

    Queue<GeneralTree<T>> cola = new Queue<>();
    cola.enqueue(this);
```

```
        int nivelActual = 0;

        while(!cola.isEmpty()) {
            int tamañoNivel = cola.size();
            System.out.println("Nivel " + nivelActual + ":");

            // Procesamos exactamente los nodos del nivel actual
            for(int i = 0; i < tamañoNivel; i++) {
                GeneralTree<T> nodoActual = cola.dequeue();
                System.out.print(nodoActual.getData() + " ");

                // Encolamos los hijos para el siguiente nivel
                for(GeneralTree<T> hijo: nodoActual.getChildren()) {
                    cola.enqueue(hijo);
                }
            }

            System.out.println(); // Nueva línea para separar niveles
            nivelActual++;
        }
    }
```

Aplicaciones de recorrido por niveles en el código:

1. **Búsqueda de un valor**: Como en el método `nivel(T dato)`, donde buscamos en qué nivel se encuentra un valor.
2. **Cálculo de características del árbol**: Como en `ancho()`, donde determinamos el nivel con máximo número de nodos.
3. **Filtrado de nodos**: Como en `numerosImparesMayoresQuePorNiveles(Integer n)` donde recolectamos nodos que cumplen ciertos criterios.

Comparación con recorridos en profundidad:

El código muestra implementaciones tanto de recorridos en profundidad (pre-orden, in-orden, post-orden) como recorrido en anchura (por niveles). La principal diferencia:

- **Recorridos en profundidad**: Usan recursión para explorar completamente un subárbol antes de pasar al siguiente.
- **Recorrido en anchura**: Usa una cola para visitar todos los nodos de un nivel antes de pasar al siguiente nivel.

Cómo se determina el nivel:

En el recorrido por niveles, podemos rastrear en qué nivel nos encontramos incrementando un contador después de procesar todos los nodos del nivel actual, como se muestra en el método `nivel(T dato)`:

```
// Cada vez que terminamos de procesar todos los nodos de un
        nivel:
level++;
```

La clave para identificar correctamente los niveles es capturar el tamaño de la cola antes de comenzar a procesar cada nivel, para saber exactamente cuántos nodos debemos sacar de la cola antes de pasar al siguiente nivel.