

# Taller 2

Alan Jesús Navarro Montes  
Gabriel Giovanni González Galindo  
Jonathan Alberto Ortiz Rodríguez.

November 3, 2014

## 1 Introducción

Los temas a considerar en este taller fueron se basaron en arboles binarios y colas, entre ellos los “Heaps” y la “LinearListPriorityQueue”. Un árbol binario es una estructura que solo tiene máximo 2 nodos, y cada nodo tiene la dirección de estos; un “Heap” es un árbol binario con la característica de que es completo, es decir todos sus niveles están llenos a excepción posiblemente del ultimo, en la que todos sus nodos están lo más a la izquierda posible; el objetivo principal de un “Heap” es jerarquizar sus elementos por nivel, así el nodo raíz es el mayor o menor de todos los sub-nodos. Así mismo el objetivo de la “LinearListPriorityQueue” es generar una estructura de tipo cola (es decir el primero que entra es el primero que sale) en la que se tiene en cuenta al igual que en el caso de “Heaps” una jerarquización de los elementos en donde, en el caso específico del taller el primer elemento que va a salir es el mayor de los que se encuentren.

## 2 Definición del Problema

1. En el primer punto se quiere generar nuevos métodos y cambiar la implementación de los recorridos “preOrderOutput”, “inOrderOutput” y “posOrderOutput” en una nueva clase “MyLBT” la cual extiende la funcionalidad de “LinkedBinaryTree”. Estos nuevos métodos son “brother” (retorna el hermano de algún elemento en el árbol), “shuffle” (cambia el contenido de los nodos pero no la estructura del árbol) y “diameter” que retorna la distancia entre 2 nodos.
2. En el segundo punto se quiere hacer una comparación de la rapidez con la que se realizan los procesos entre “MaxHeap” Y “LinearListPriorityQueue”.
3. En el punto 3 se busca que dada una lista de números ordenados descendientemente se busca la k-esima suma más grande sin hacer las combinaciones posibles.

## 3 Código Importante

1. Primer Punto.

- Función principal dentro del método **brother()**

```

static <T> T thebro(BinaryTreeNode<T> t, T e){
    //only check the nodes with 2 childs
    if(t.rightChild==null || t.leftChild==null) {
        return null;
    }
    // I found his brother
    if(t.leftChild.element.equals(e))
        return t.rightChild.element;
    else if(t.rightChild.element.equals(e)){
        return t.leftChild.element;
    }

    T e1=thebro(t.leftChild,e);
    T e2=thebro(t.rightChild,e);
    if(e1==null){//if e1 ==null means that the node is probably in the root
        return e2;
    }
    if(e2==null){//if e2 ==null means that the node is probably in the root
        return e1;
    }
    //not found
    return null;
}

```

- Función principal para el método **diameter()**

```

static <T> int thediameter(BinaryTreeNode<T> t){
    int d=0;
    if(t==null)
        return 0;
    //calculo de la Altura
    int lh=theHeight(t.leftChild);
    int rh=theHeight(t.rightChild);
    //se calcula el radio izquierdo y derecho del diametro DE CADA NODO me
    int ld=thediameter(t.leftChild);
    int rd=thediameter(t.rightChild);
    int maxd=0;
    //total, obtiene la "distancia" mas larga en total si pasa por el root,
    //max retorna la distancia que se convierte en diametro en caso contrar
    //luego retorno d que es el maximo entre max y total que son diametros
    int total=lh+rh+1;
    if(ld>=rd){
        maxd=ld;
    }
    else{
        maxd=rd;
    }
    if(total>=maxd){
        d=total;
    }
}

```

```

else{
    d=maxd;
}
return d;
}

```

- Función principal para el método *shuffle()*

```

public void shuffle(){
    BinaryTreeNode<T> cur;
    ArrayQueue<BinaryTreeNode<T>> q = new ArrayQueue<>();
    ArrayLinearList<T> l = new ArrayLinearList<>();
    q.put(root);
    int count=0;
    while(!q.isEmpty()){
        cur=q.remove();
        l.add(count, cur.element);
        if(cur.leftChild!=null)q.put(cur.leftChild);
        if(cur.rightChild!=null)q.put(cur.rightChild);
        count++;
    }
    BinaryTreeNode<T> curn;
    q.put(root);
    while(!q.isEmpty()){
        curn=q.remove();
        int i=0;
        do{
            i=Math.abs(new Random( new Date().getTime()).nextInt())
        }
        while(l.get(i).equals(curn.element));
        curn.element=l.get(i);
        l.remove(i);
        if(curn.leftChild!=null)q.put(curn.leftChild);
        if(curn.rightChild!=null)q.put(curn.rightChild);
        count--;
    }
}

```

- Modificaciones para convertir los métodos en iterativos *preOrder-Output*, *inOrderOutput* y *posOrderOutput*

```

public ArrayList<T>iPreOrder(BinaryTreeNode<T>t){
    ArrayList<T>arr=new ArrayList<>();
    ArrayStack<BinaryTreeNode<T>> s = new
        ArrayStack<BinaryTreeNode<T>>();
    s.push(root);
    while(!s.isEmpty()){
        BinaryTreeNode<T> x = s.pop();
        arr.add(x.element);
        if(x.rightChild != null){
            s.push(x.rightChild);
        }
    }
}

```

```

        }
        if(x.leftChild != null){
            s.push(x.leftChild);
        }
    }
    return arr;
}

public ArrayList<T> iInOrder(BinaryTreeNode<T>t){
    ArrayList<T>arr=new ArrayList<>();
    ArrayStack<BinaryTreeNode<T>>s=new ArrayStack<>();
    BinaryTreeNode<T>curr=root;
    while(!s.isEmpty() || curr != null){
        if(curr != null){
            s.push(curr);
            curr = curr.leftChild;
        }else{
            BinaryTreeNode<T>x = s.pop();
            arr.add(x.element);
            curr = x.rightChild;
        }
    }
    return arr;
}

public ArrayList<T> iPostOrder(BinaryTreeNode<T>t){
    ArrayList<T>arr=new ArrayList<>();
    if(root==null)
        return arr;
    ArrayStack<BinaryTreeNode<T>>s=new ArrayStack<>();
    s.push(t);
    BinaryTreeNode<T> prev = null;
    while(!s.isEmpty()){
        BinaryTreeNode<T> curr = s.peek();
        if(prev == null || prev.leftChild == curr ||
            prev.rightChild == curr){
            if(curr.leftChild != null){
                s.push(curr.leftChild);
            }
            else if(curr.rightChild != null){
                s.push(curr.rightChild);
            }
            else{
                s.pop();
                arr.add(curr.element);
            }
        }
        else if(curr.leftChild == prev){
            if(curr.rightChild != null){
                s.push(curr.rightChild);
            }

```

```

        }
        else{
            s.pop();
            arr.add(curr.element);
        }
    }
    else if(curr.rightChild == prev){
        s.pop();
        arr.add(curr.element);
    }
    prev = curr;
}
return arr;
}

```

2. Segundo Punto.

- **Main** de *LinearListPriorityQueue*

```

public static void main( String[] args ){
    LinearListPriorityQueue< Integer > q
        = new LinearListPriorityQueue<>( 3 );
    ////////////////////////////////// FASE 1 //////////////////////////////////
    int j=0;
    Random ale= new Random();
    long time = System.currentTimeMillis();
    for (int i=0; i<100000;i++){
        j=(int) (ale.nextDouble() * 100000000+1);
        q.put(new Integer(j));
    }
    // EN ESTE CICLO SE HACE LA INSERCIÓN DE LOS 100000
    DATOS ALEATORIOS
    }
    time = System.currentTimeMillis() - time;
    System.out.println("El tiempo de la Fase 1 en
microsegundos fue:" + time);
    ////////////////////////////////// FASE 2 //////////////////////////////////
    long time2 = System.currentTimeMillis();
    for (int i=0; i<50000;i++){
        q.removeMax();
        q.getMax();
    }
    // EN ESTE CICLO SE HACE LA ITERACIÓN 50000 VECES DE
    LOS METODOS getMax() y removeMax()
    }
    time2 = System.currentTimeMillis() - time2;
    System.out.println("El tiempo de la Fase 2 en
microsegundos fue:" + time2);
}

```

- **Main** de *MaxHeap*

```

public static void main( String[] args )

```

```

{
    MaxHeap<Integer> h = new MaxHeap<>( 4 );
    ////////////////////////////////// FASE 1 //////////////////////////////////
    int j=0;
    Random ale= new Random();
    long time = System.currentTimeMillis();
    for (int i=0; i<100000;i++){
        j=(int) (ale.nextDouble() * 100000000+1);
        h.put(new Integer(j));
    }
    // EN ESTE CICLO SE HACE LA INSERCIÓN DE LOS 100000
    DATOS ALEATORIOS
    }
    time = System.currentTimeMillis() - time;
    System.out.println("El tiempo de la Fase 1 en
    .....microsegundos fue: " + time);
    ////////////////////////////////// FASE 2 //////////////////////////////////
    long time2 = System.currentTimeMillis();
    for (int i=0; i<50000;i++){
        h.removeMax();
        h.getMax();
    }
    // EN ESTE CICLO SE HACE LA ITERACIÓN 50000 VECES DE
    LOS METODOS getMax() y removeMax()
    }
    time2 = System.currentTimeMillis() - time2;
    System.out.println("El tiempo de la Fase 2 en
    .....microsegundos fue: " + time2);
}

```

3. Tercer Punto.

## 4 Pantallazos

- Primer Punto

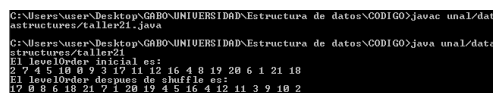


```

C:\Users\Suser\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>javac unal\data
estructuras/taller21.java
C:\Users\Suser\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/taller21
the brother of 1 is null
the brother of 19 is 8

```

Figure 1: Ejecución del método *brother()*.



```

C:\Users\Suser\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>javac unal/data
estructuras/taller21.java
C:\Users\Suser\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/taller21
El levelOrder inicial es:
2 7 4 5 10 0 9 3 17 11 12 16 4 8 19 20 6 1 21 18
El levelOrder despues de shuffle es:
17 0 8 6 18 21 7 1 20 19 4 5 16 4 12 11 3 9 10 2

```

Figure 2: Ejecución del método *shuffle()*.

```

C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>javac unal\data
estructuras/taller21.java
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/taller21
Iterative preorder:
12, 9, 5, 8, 19, 20, 10, 11, 6, 12, 1, 4, 0, 16, 4, 9, 8, 21, 19, 181
Recursive preorder:
2 7 5 3 17 20 10 11 6 12 1 4 0 16 4 9 8 21 19 18
Iterative inorder:
13, 5, 20, 17, 7, 6, 11, 10, 12, 1, 2, 16, 0, 4, 4, 8, 21, 9, 18, 191
Recursive inorder:
3 5 20 17 7 6 11 10 12 1 2 16 0 4 4 8 21 9 18 19
Iterative postorder:
13, 20, 17, 5, 6, 11, 1, 12, 10, 7, 16, 4, 0, 21, 0, 10, 19, 9, 4, 21
Recursive postorder:
3 20 17 5 6 11 1 12 10 7 16 4 0 21 0 18 19 9 4 2

```

Figure 3: Ejecución de los métodos *inOrder()*, *preOrder()*, *posOrder()* de manera iterativa.

```

C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>javac unal/data
estructuras/taller21.java
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/taller21
level order output is
5 6 9 7 1 4 2 0 3
diameter of 1st tree: 9
5 6 9 7 1 4 2 0 3
diameter of 2nd tree: 7

```

Figure 4: Ejecución del método *diameter()*.

- Segundo Punto

```

C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 43
El tiempo de la Fase 2 en micro segundos fue: 50600
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 53770
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 53459
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 51928
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 50235
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 42
El tiempo de la Fase 2 en micro segundos fue: 49288
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 49221
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 43
El tiempo de la Fase 2 en micro segundos fue: 50730
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 40
El tiempo de la Fase 2 en micro segundos fue: 49799
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO>java unal/data
estructuras/LinearListPriorityQueue
El tiempo de la Fase 1 en micro segundos fue: 50
El tiempo de la Fase 2 en micro segundos fue: 50300

```

Figure 5: Exposición de los tiempos gastados en la adición de los 100000 datos y la ejecución 50000 veces de los métodos *getMax()* y *removeMax()* en *LinearListPriorityQueue*.

```

C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 55
El tiempo de la Fase 2 en micro segundos fue: 70
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 55
El tiempo de la Fase 2 en micro segundos fue: 67
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 58
El tiempo de la Fase 2 en micro segundos fue: 65
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 52
El tiempo de la Fase 2 en micro segundos fue: 65
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 55
El tiempo de la Fase 2 en micro segundos fue: 65
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 65
El tiempo de la Fase 2 en micro segundos fue: 73
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 55
El tiempo de la Fase 2 en micro segundos fue: 68
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 57
El tiempo de la Fase 2 en micro segundos fue: 77
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 55
El tiempo de la Fase 2 en micro segundos fue: 67
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\MaxHeap
El tiempo de la Fase 1 en micro segundos fue: 68
El tiempo de la Fase 2 en micro segundos fue: 68

```

Figure 6: Exposición de los tiempos gastados en la adición de los 100000 datos y la ejecución 50000 veces de los métodos *getMax()* y *removeMax()* en *MaxHeap*.

- Tercer Punto

```

C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\Ksumpair.java
C:\Users\User\Desktop\GABO\UNIVERSIDAD\Estructura de datos\CODIGO\java unal\data
structure\Ksumpair
The 8 elements are [ 28, 21, 19, 14, 17, 12, 5, 10 ]
The 5 elements are [ 28, 21, 19, 14, 17 ]
The 3 suma es :19
The 9 elements are [ 28, 21, 19, 14, 17, 12, 5, 10, 3 ]
The novena suma es :2

```

Figure 7: Búsqueda de la k-esima suma en 3 arboles distintos. *getMax()* y *removeMax()* en *MaxHeap*.

## 5 Que Aprendimos

1. Para el primer punto pudimos observar, analizar y aprender la manera de usar la recursividad, y la iteración en los diferentes métodos realizados para el desarrollo del taller.
2. Para el segundo punto obtuvimos los siguientes resultados

Estructura/Fase	MaxHeap	PROMEDIO	LinearListPriorityQueue	PROMEDIO	TOTAL
Fase 1	55	56,7	43	41,8	98,5
	55		40		
	58		40		
	52		40		
	55		40		
	65		42		
	55		40		
	57		43		
	55		40		
	60		50		
Fase 2	70	70,5	50608	50935,4	51005,9
	67		53778		
	65		53459		
	65		51928		
	65		50235		
	73		49288		
	68		49221		
	97		50730		
	67		49799		
	68		50308		
TOTAL		127,2		50977,2	

Figure 8: Resultados ejecución.



Con lo cual podemos concluir que el "*MaxHeap*" es más rápido para hacer los procesos.

- Para el tercer punto Dado que las listas están en orden descendente pudimos observar que:
  - La posición 0,0 es siempre el elemento mayor del heap
  - La posición 2,2 es siempre el elemento menor del heap, en el caso de N sería la posición N,N
  - La posición 0,1 y la 1,0 son :
    - \* Mayores que 0,2 y 2,0 respectivamente y por consiguiente de los demás sub índices(11,12,21,22)
    - \* La posición 01 es mayor , o es menor o son iguales respecto a su homologo el 10 luego para saber quién es mi segunda mayor suma debo añadir al heap tanto al 10 como al 01 por lo mencionado anteriormente.