# Práctica 1: Árboles generales

## Normas:

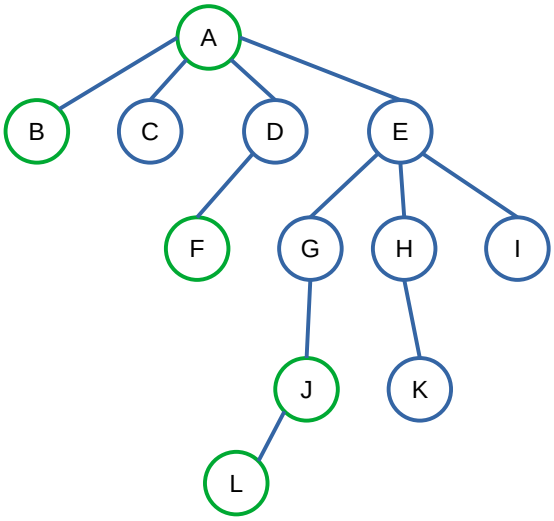
- Cada alumno debe realizar los ejercicios de manera individual, aunque pueden compartir información oral. Existe un detector anticopias.
- Este enunciado va acompañado de un fichero ZIP que contiene el código necesario para hacer la práctica. Los test que acompañan a las prácticas son orientativos y pueden ser ampliados en la fase de evaluación.

## Ejercicio 1: Iteradores sobre árboles

Para aumentar la funcionalidad de los árboles se desea disponer de diferentes tipos de recorridos sobre ellos. Para ello se propone que implemente las clases `LeafIterator`, `BreadthFirstTreeIterator`. El esqueleto de las clases se proporcionan en el proyecto.

## Ejercicio 2: Vistas laterales sobre árboles

Se desea implementar métodos para poder "visualizar" un árbol desde su lado izquierdo y desde su lado derecho, estos métodos reciben un árbol y devuelven una lista con los elementos del árbol que se visualizarían al mirarlo desde la izquierda o desde la derecha. Estos métodos están en la clase `MoreFunctionality`. Dado el siguiente árbol




el método `Leftview` devolvería una lista con los elementos: A, B, F, J y L. Si la vista es desde el lado derecho, `Rigthview` la lista estará formada por los elementos: A, E, I, K y L.

## Ejercicio 3: Feendwick Tree

Consideremos el siguiente problema para entender el árbol indexado binario (o Fendwick Tree). Tenemos un array arr[0 . . . n-1] y queremos:

1. Calcular la suma de los primeros i elementos del array.
2. Modificar el valor de un elemento determinado del array arr[i] = x donde 0 <= i <= n-1.

Pero además deseamos que los cálculos de las sumas parciales tengan una complejidad de O(logn) => necesitamos un Fendwick Tree.

**¿Cómo funciona el Fendwick Tree?**
La idea se basa en el hecho de que todos los enteros positivos pueden representarse como la suma de potencias de 2. Por ejemplo, 19 puede representarse como 16 + 2 + 1 (10000 + 00010 + 00001). Cada nodo del Fendwick Tree almacena la suma de n elementos donde n es una potencia de 2.

Dado un array de enteros (cada entero ocupará una posición del array):

| ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr | 2 | 1 | 1 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Queremos calcular sus sumas parciales y que los elementos del array (arr) se puedan modificar sin que las sumas parciales tengan una complejidad alta a la hora de recalcularse. Entonces construimos un Fendwick Tree (BITree) que mantiene las sumas parciales de partes del array original. Para ello necesitaremos:

- BITree[0] es un nodo ficticio, en el árbol se indica como nodo dummy.
- BITree[y] es el padre de BITree[x], si y sólo si y puede obtenerse eliminando el último bit establecido de la representación binaria de x, es decir y = x - (x & (-x)).
- El nodo hijo BITree[x] del nodo BITree[y] **almacena la suma de los elementos entre y(inclusive) y x(exclusive): arr[y,...,x).**

Entonces para el array de arriba (arr), su BITree será:
Por simplicidad a la operación x&(-x) lo que hace es obtener el bit a 1 menos significativo de x (no es necesario que hagamos la transformación a binario, porque Java lo hace internamente). Pero lo que hace es: x = 5 → y = (101) – (101 & (011)) = 100; y = 4

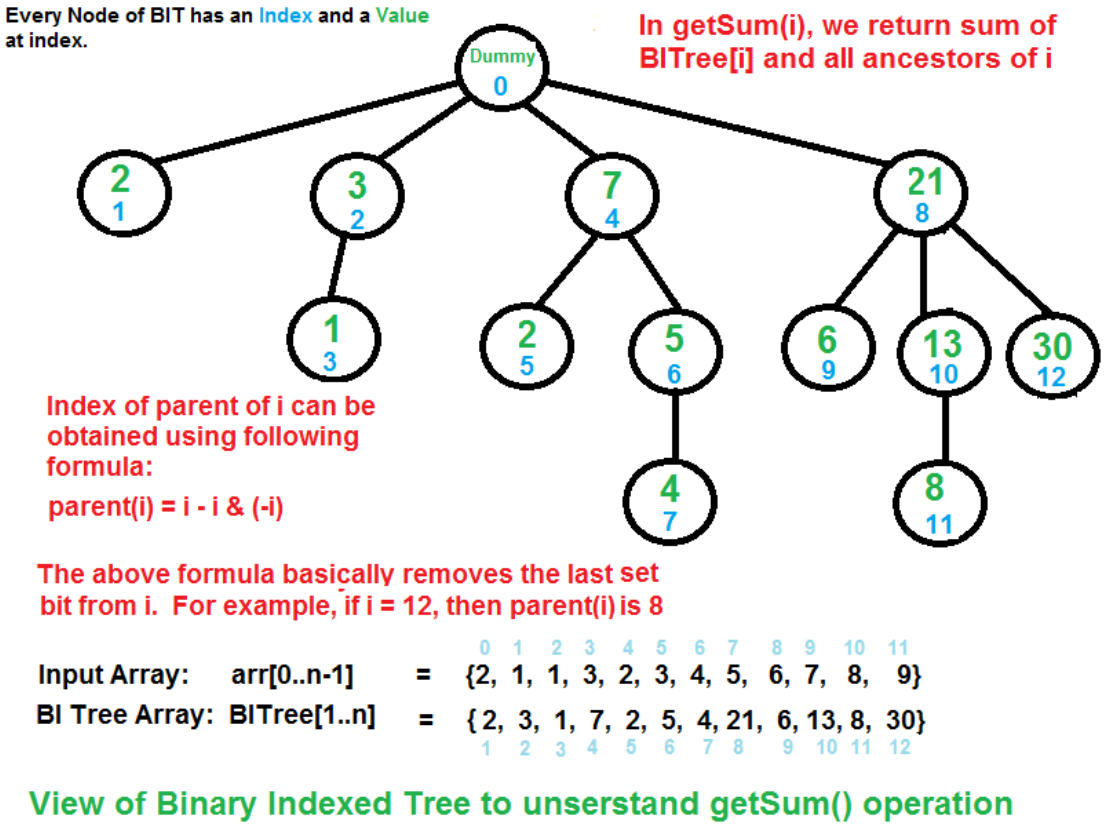
El índice 0 es un nodo ficticio, de manera que no lo representamos.
El padre del índice 1 será = 0, se suma el intervalo del arr [0,1) = 2
El padre del índice 2 será = 0, se suma el intervalo del arr [0,2) = 3 (2+1)
El padre del índice 3 será = 2, se suma el intervalo del arr [2,3) = 1
El padre del índice 4 será = 0, se suma el intervalo del arr [0,4) = 7 (2+1+1)
El padre del índice 5 será = 4, se suma el intervalo del arr [4,5) = 2
El padre del índice 6 será = 4, se suma el intervalo del arr [4,6) = 5 (2+3)
El padre del índice 7 será = 6, se suma el intervalo del arr [6,7) = 4
El padre del índice 8 será = 0, se suma el intervalo del arr [0,8) = 21 (2+1+1+3+2+3+4+5)
El padre del índice 9 será = 8, se suma el intervalo del arr [8,9) = 6
El padre del índice 10 será = 8, se suma el intervalo del arr [8,10) = 13 (6+7)
El padre del índice 11 será = 10, se suma el intervalo del arr [10,11) = 8
El padre del índice 12 será = 8, se suma el intervalo del arr [8,12) = 30 (6+7+8+9)
Quedando el BITree:

| ind | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BITree | 2 | 3 | 1 | 7 | 2 | 5 | 4 | 21 | 6 | 13 | 8 | 30 |

Las operaciones que nos interesan del Fendwick Tree son getSum(x) que dado un x nos devuelve la suma parcial sobre el arr desde 0 a x incluido. Para ello, previamente tendremos calculado nuestro BITree sobre el array original. El algoritmo de getSum será el siguiente:

- inicializamos el valor que vamos a devolver a cero, y actualizamos el índice x a x+1 (porque nuestro Fendwick Tree tiene en 0 el nodo dummy).
- Mientras el valor de índice sera mayor que cero, añadimos al valor suma el valor almacenado en BiTrre[index] y actualizamos index al valor de su padre => index = index-(index&(-index))
- Al salir del bucle ya tenemos el valor de la suma solicitada.

En la figura de abajo, podemos ver la representación del ejemplo inicial con el Fendwick Tree representado como un árbol.




View of Binary Indexed Tree to unserstand getSum() operation

Si sobre nuestro ejemplo llamásemos a getSum(7) obtendríamos 21, que en este caso se obtendría directamente. Otro posible ejemplo sería obtener la suma de los primeros 12 elementos (en nuestro ejemplo todo el array inicial), como se puede ver se obtendría mediante la suma de los últimos 4 elementos (del 9 al 12) más la suma de 8 elementos (del 1 al 8).

La otra operación que debemos implementar es la operación update(x, val). Exteriormente el usuario tendrá pensará que se ha actualizado el array original tal que arr[index] += val, aunque en realidad solo se modifica el Fendwick Tree (BITree). El algoritmo que se sigue es el siguiente:

- Inicialice el índice actual como x+1.
- Mientras el índice sea menor o igual a n:
  - Suma val a BITree[índice]

- Ir al siguiente elemento de BITree[índice]. El siguiente elemento se puede obtener incrementando el último bit establecido del índice actual, es decir, índice = índice + (índice & (-índice))

Dado nuestro ejemplo inicial, si llamásemos a update(5,3) nuestro Fendwick Tree solo tendrá que modificar las posiciones 6 y 8 (que son las que se calculaban sumando la posición 5 del array original).

Quedando el BITree:

| ind | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BITree | 2 | 3 | 1 | 7 | 2 | **8** | 4 | **24** | 6 | 13 | 8 | 30 |

Implemente un Fendwick Tree que posea las siguientes operaciones:

`public FendTree (int[] array);` Que recibe un array de enteros con el que se creará el Fendwick Tree

`public int getSum(int index);` Que recibe un índice sobre el array y calcula la suma parcial desde 0 hasta index

`public void upDate(int index, int val);` Que recibe un índice y el valor que hay que sumarle a dicha posición del array, esta operación actualizará el Fendwick Tree.