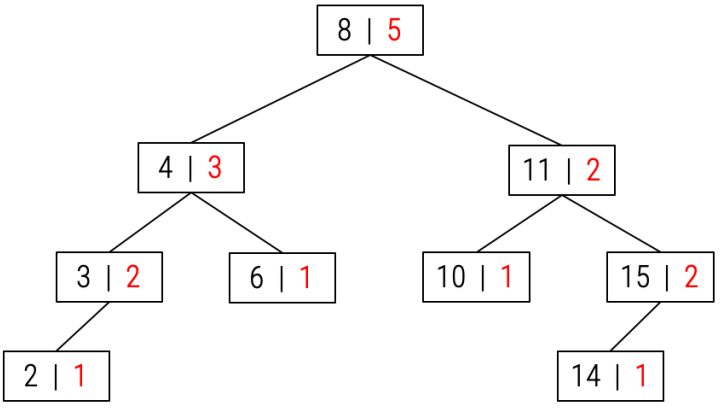
1.1 k-ésimo menor elemento

El objetivo de este ejercicio es extender el TAD de los conjuntos con una nueva operación `kesimo` que, dado un entero *k*, devuelva (si existe) el *k*-ésimo menor elemento del conjunto, es decir, el elemento que ocuparía la posición *k* si los elementos del conjunto se ordenaran. Por ejemplo, si *k* es 1, devolverá el menor elemento del conjunto; si *k* es 2, devolverá el segundo menor elemento del conjunto; y si *k* es igual al número de elementos del conjunto, devolverá su mayor elemento.

Con nuestra implementación actual solamente podríamos implementar esta operación con un coste lineal en el número de elementos del conjunto, en el caso peor. Para poder conseguir una implementación más eficiente, antes vamos a modificar la representación de los conjuntos mediante árboles AVL, añadiendo a cada nodo del árbol un nuevo atributo, `tam_i`, que almacene el número de nodos en el hijo izquierdo más 1. Es decir, ese valor dirá qué posición ocuparía la raíz del subárbol si sus elementos se ordenaran. En la siguiente figura, en cada nodo del árbol AVL aparece en negro el elemento en ese nodo y en rojo el valor del nuevo atributo `tam_i`:



Modificad la implementación de la clase `Set` (fichero `TreeSet_AVL.h`) para añadir este nuevo atributo y haced los cambios necesarios para que las funciones que modifican los árboles mantengan coherente el valor de dicho atributo. Para simplificar, podéis suponer que no hay operación de eliminación.

Solución: (Escribid aquí las explicaciones necesarias para contar de manera comprensible vuestra solución. También podéis añadir dibujos, o fotos de dibujos, y el código que consideréis oportuno, pero no peguéis el código completo de la clase `Set`, solo lo modificado. Extended el espacio al que haga falta.)

```
struct TreeNode;
using Link = TreeNode*;
struct TreeNode {
 T elem;
 Link iz, dr;
 int altura;
 int numNodosIz;
 TreeNode(T const& e, Link i = nullptr, Link d = nullptr,
  int alt = 1, int numNodosIzd = 1) : elem(e), iz(i), dr(d), altura(alt), numNodosIz(numNodosIzd) {}
};
```

```
bool inserta(T const& e, Link& a) {
 bool crece;
 if (a == nullptr) { // se inserta el nuevo elemento e
  a = new TreeNode(e);
  ++nelems;
  crece = true;
 }
 else if (menor(e, a->elem)) {
  crece = inserta(e, a->iz);
  if (crece) {
   a->numNodosIz++;

   reequilibraDer(a);
  }
 }...
}
```

```
void rotaDer(Link& r2) {
 Link r1 = r2->iz;
 r2->iz = r1->dr;
 r1->dr = r2;
 r2->altura = std::max(altura(r2->iz), altura(r2->dr)) + 1;
 r1->altura = std::max(altura(r1->iz), altura(r1->dr)) + 1;
 r2->numNodosIz -= r1->numNodosIz;


 r2 = r1;
}

void rotaIzq(Link& r1) {
 Link r2 = r1->dr;
 r1->dr = r2->iz;
 r2->iz = r1;
 r1->altura = std::max(altura(r1->iz), altura(r1->dr)) + 1;
 r2->altura = std::max(altura(r2->iz), altura(r2->dr)) + 1;
 r2->numNodosIz += r1->numNodosIz;
 r1 = r2;
}
```

A continuación, extended la clase `Set` con un nuevo método

```
T const& kesimo(int k) const
```

que devuelva el *k*-ésimo menor elemento del conjunto, si existe. Si no existe, la operación lanzará una excepción. El coste de esta nueva operación debe ser logarítmico respecto al número de elementos en el conjunto.

Solución: (Escribid aquí las explicaciones necesarias para contar de manera comprensible vuestra solución. Justificad el coste del nuevo método.)

```
T const & kesimoAux(Link a, int k) const {
  if (a == nullptr)
   return -1;
  if (k < a->numNodosIz)
   return kesimoAux(a->iz, k);
  else if (k > a->numNodosIz)
   return  kesimoAux(a->dr, k - a->numNodosIz);
  else return a->elem;


 }
```

```
T const& kesimo(int k) const {
 if (k > this->size()) throw ENumElemsMenc

 return kesimoAux(raiz, k);


}
```

Podéis entregar vuestra solución en el **Problema 2** del juez automático, aunque durante la realización del ejercicio el juez no tendrá más casos de prueba que los que aparecen aquí abajo.

Número de envío:

Descripción de la entrada

La entrada está formada por diversos casos de prueba. Cada caso ocupa cuatro líneas. En la primera aparece el número *N* de valores (entre 1 y 50.000), no necesariamente distintos, a insertar en el conjunto. A continuación, en una misma línea, aparecen esos valores (números enteros entre 1 y 1.000.000), separados por espacios, en el orden en que deben ser insertados. En la siguiente línea aparece el número *M* de elementos que se van a consultar (entre 1 y *N*). Y en la última línea aparecen, separadas por espacios, las *M* posiciones (ordinales) de los elementos a consultar (números entre 1 y *N*).
La entrada termina cuando un árbol no tiene elementos (*N* es 0).

Descripción de la salida

Para cada caso de prueba se escribirán, en líneas separadas, los valores consultados, si existen. Si no existe la posición consultada, se escribirá `??` en su lugar.
Tras procesar cada caso se escribirá una línea más con tres guiones, `---`.

Entrada de ejemplo

```
4
15 20 25 30
2
1 3
5
16 8 4 4 32
3
2 4 5
0
```

Salida de ejemplo

```
15
25
---
8
32
??
---
```