2. Considérese un árbol binario de búsqueda BST<T> instanciado para objetos del tipo T. Impleméntense las dos operaciones siguientes: Sin usar iteradores

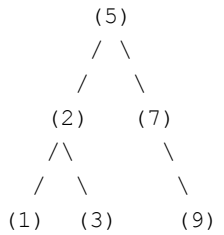
- pair<iterator,iterator> rango(T inicial, T final) const, que devuelve un par de iteradores del BST entre los que están contenidos todos los valores en el rango [inicial, final].
- bool operator==(const BST<T> & bst) const, que indica si dos BST son iguales o no.

Este ejercicio sólo tiene sentido si el recorrido que se debe hacer con los iteradores es inorden.

Por qué?

Porque si fuera recorrido preorden o postorden, nos tendríamos que saltar nodos durante el operator++ para mostrar únicamente los que hay en el intervalo [inicio,final].

Ejemplo:



Los iteradores que se devolverían para [2,7] serían 2 y 9 (siguiente al último).
En recorrido preorden, después del 2 se recorrería el 1, que queda fuera del intervalo. No puede ser.

Los iteradores que se devolverían para [3,7] serían 3 y 9 (siguiente al último).
En recorrido postorden, después del 3 se recorrería el 2, que queda fuera del intervalo. No puede ser.

Supondremos que el BST, que está fundamentado en un bintree, tiene un miembro llamado arbolb que es este bintree.

```
// Esta función te devuelve en n el nodo si lo hay, y en np el padre.
template <typename T>
bool BST<T>::encontrar(typename bintree<T>::node& n,
                       typename bintree<T>::node& np, const T& clave) const {
  bool enc=false;

  n=arbolb.root();
  np=bintree<T>::node();
```

```
  while (!n.null() && !enc) {
    if (clave==*n)
      enc=true;
    else {
      np=n;
      if (clave<*n)
        n=n.left();
      else
        n=n.right();
    }
  }
  return enc;
}

template <typename T>
pair<typename BST<T>::iterator,typename BST<T>::iterator> BST<T>::rango(
                                                        T inicial, T final) const {
  bintree<T>::node n,aux;
  pair<iterator,iterator> p;

  //Para encontrar el inicial...

  //Lo buscamos, si lo encontramos es ese.
  if (encontrar(n,aux,inicial))
    iterator it1(n);
  else {
    //Si no, vamos subiendo por los padres hasta que el nodo actual sea mayor
    //  que lo que buscamos. Ver anotación (1) al final.
    while (!aux.null() && *aux<inicial)
      aux=aux.parent();
    iterator it1(aux);
  }


  //Para encontrar el final...

  //Lo buscamos. Si lo encontramos, debemos devolver el siguiente! (recordad
  //cómo funcionan los iteradores, se devuelve el siguiente para saber cuándo
  //  parar)
  encontrar(n,aux,final);

  //Tanto si lo encontramos como si no,
  //Subiríamos por los padres hasta que el nodo actual sea mayor que lo que
  //  buscamos. Ver anotación (2) al final.
  while (!aux.null() && *aux<=final)
    aux=aux.parent();
  iterator it2(aux);


   p.first=it1;
```

```
   p.second=it2;
   return p;
}


//Se usa el operador de comparación del árbol binario.
template <typename T>
inline bool BST<T>::operator==(const BST<T> & bst) const {
  return arbolb==bst.arbolb;
}
```

(1) Por qué esto? Siempre que no encontremos el exacto nos va a interesar el que es un poquito mayor que el que buscamos.

Si el nodo en cuestión no está, se puede haber bajado a él desde su padre, bien por la izquierda o bien por la derecha.

Si bajamos por la izquierda, el padre será mayor, y será justamente el inmediato más mayor a este nodo. Con este código funciona.

Si bajamos por la derecha, el padre será menor. A ese padre podremos haber llegado por la izquierda o por la derecha. Se repite recursivamente el razonamiento. El primer padre mayor que encontremos será el siguiente.

(2) Por qué esto? Vamos a querer SIEMPRE el que es un poquito mayor que el que buscamos, nunca el exácto! Así se nos devolverá el siguiente y podremos utilizar final como iterador de end.

El cómo llegamos al que es un poquito mayor ya está explicado en (1), tenéis que tener en cuenta que la condición de parada aquí es *aux<=final.