Hoy:
pair, tuples
BST
Operaciones BST
AVL
Red-Black trees
Ejercicios

```cpp
#ifndef PAIR_H_
#define PAIR_H_

#include "StartConv.h"


// Class (more like a struct) that stores a pair of objects.
template <class Type1, class Type2>
class pair
{
  public:
    Type1 first;
    Type2 second;

    pair( const Type1 & f = Type1( ), const Type2 & s = Type2( ) ) :
      first( f ), second( s ) { }
};

#include "EndConv.h"

#endif
```

```cpp
#include <iostream>
#include <tuple>

using namespace std;

auto fact(int i) {
    return make_pair((double) i, i);
}


int main()
{


    auto pairExample = std::make_pair(2, 3);
    auto pairReturned = fact(1);
    auto tupleExample = tuple<int, double, string>();
    auto& tupleExampleWithoutFirst = tupleExample._Get_rest();


    auto value = get<1>(tupleExample);
    std::cout << "Hello World!  " << pairReturned.first << "  \n";
}
```

Binary Search Tree

```
template <class Comparable>
class BinaryNode
{
    Comparable  element;
    BinaryNode *left;
    BinaryNode *right;
    int size;

    BinaryNode( const Comparable & theElement, BinaryNode *lt,
                BinaryNode *rt, int sz = 1 )
      : element( theElement ), left( lt ), right( rt ), size( sz ) { }

    friend class BinarySearchTree<Comparable>;
    friend class BinarySearchTreeWithRank<Comparable>;
};
```

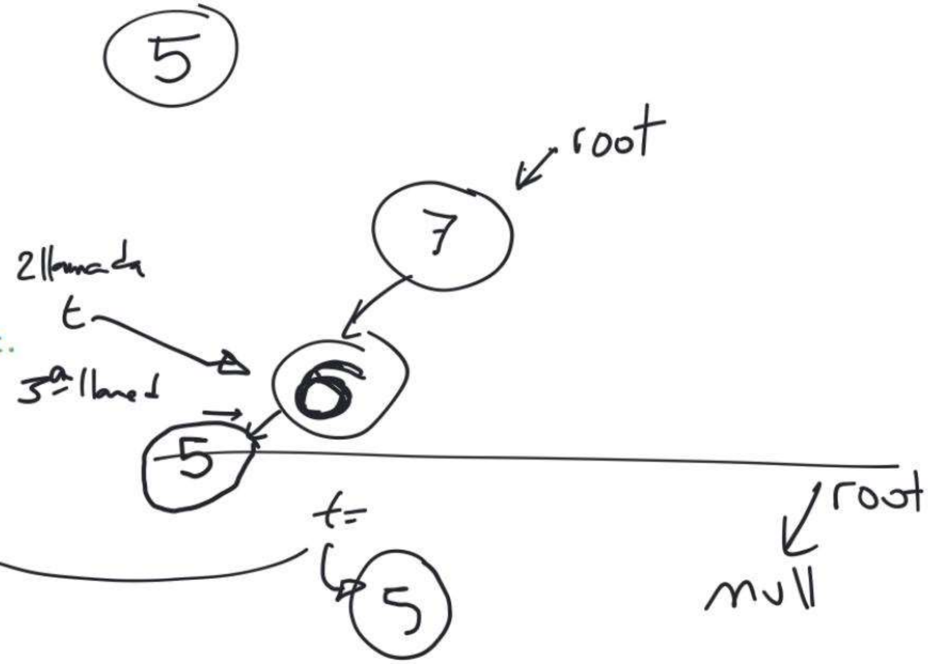
```cpp
template <class Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    virtual ~BinarySearchTree( );

    Cref<Comparable> findMin( ) const;
    Cref<Comparable> findMax( ) const;
    Cref<Comparable> find( const Comparable & x ) const;
    bool isEmpty( ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void remove( const Comparable & x );
    void removeMin( );

    const BinarySearchTree & operator=( const BinarySearchTree & rhs );

    typedef BinaryNode<Comparable> Node;
```

```
protected:
  Node *root;

  Cref<Comparable> elementAt( Node *t ) const;
  virtual void insert( const Comparable & x, Node * & t ) const;
  virtual void remove( const Comparable & x, Node * & t ) const;
  virtual void removeMin( Node * & t ) const;
  Node * findMin( Node *t ) const;
  Node * findMax( Node *t ) const;
  Node * find( const Comparable & x, Node *t ) const;
  void makeEmpty( Node * & t ) const;
  Node * clone( Node *t ) const;
};
```

```cpp
// Insert x into the tree;
// Throws DuplicateItemException if x is already there.
template <class Comparable>
void BinarySearchTree<Comparable>::insert( const Comparable & x )
{
    insert( x, root );
}
```

```cpp
// Internal method to wrap the element field in node t inside a Cref object.
template <class Comparable>
Cref<Comparable> BinarySearchTree<Comparable>::elementAt( Node *t ) const
{
    if( t == NULL )
        return Cref<Comparable>( );
    else
        return Cref<Comparable>( t->element );
}
```

```cpp
// Internal method to insert into a subtree.
// x is the item to insert.
// t is the node that roots the tree.
// Set the new root.
// Throw DuplicateItemException if x is already in t.
template <class Comparable>
void BinarySearchTree<Comparable>::
insert( const Comparable & x, Node * & t ) const
{
    if( t == NULL )
        t = new Node( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        throw DuplicateItemException( );
}
```

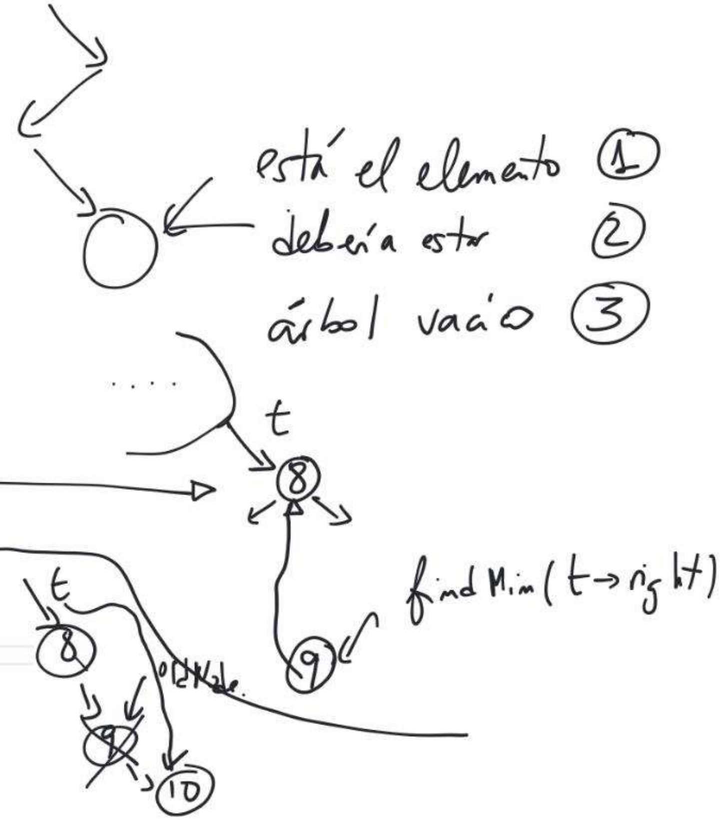


Bst < Algo>   Algo sólo le pido <

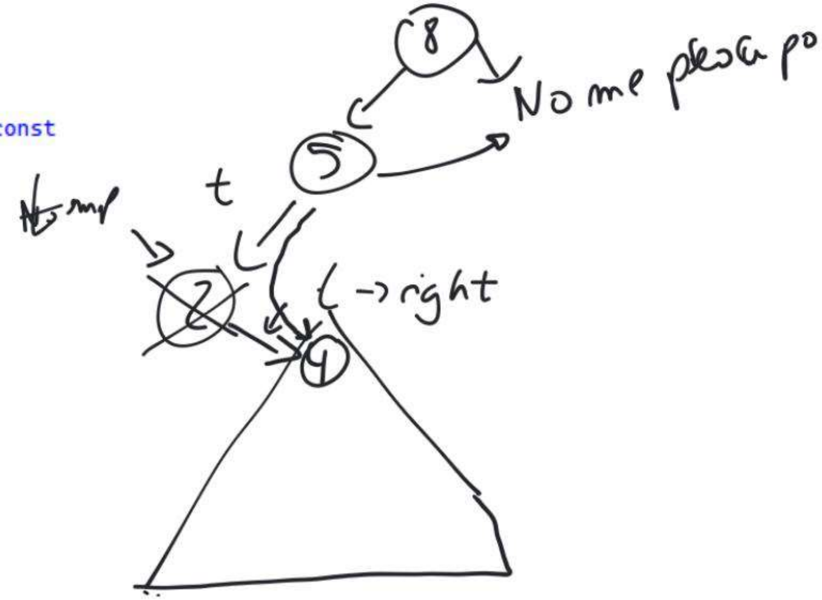
```cpp
// Remove x from the tree.
// Throws ItemNotFoundException if x is not in the tree.
template <class Comparable>
void BinarySearchTree<Comparable>::remove( const Comparable & x )
{
    remove( x, root );
}

// Remove minimum item from the tree.
// Throws UnderflowException if tree is empty.
template <class Comparable>
void BinarySearchTree<Comparable>::removeMin( )
{
    removeMin( root );
}


// Return the smallest item in the tree wrapped in a Cref object.
template <class Comparable>
Cref<Comparable> BinarySearchTree<Comparable>::findMin( ) const
{
    return elementAt( findMin( root ) );
}
```

```cpp
// Internal method to remove from a subtree.
// x is the item to remove.
// t is the node that roots the tree.
// Set the new root.
// Throw ItemNotFoundException is x is not in t.
template <class Comparable>
void BinarySearchTree<Comparable>::
remove( const Comparable & x, Node * & t ) const
{
    if( t == NULL )
        throw ItemNotFoundException( );
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        removeMin( t->right );                     // Remove minimum
    }
    else
    {
        BinaryNode<Comparable> *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;  // Reroot t
        delete oldNode;                                // delete old root
    }
}
```

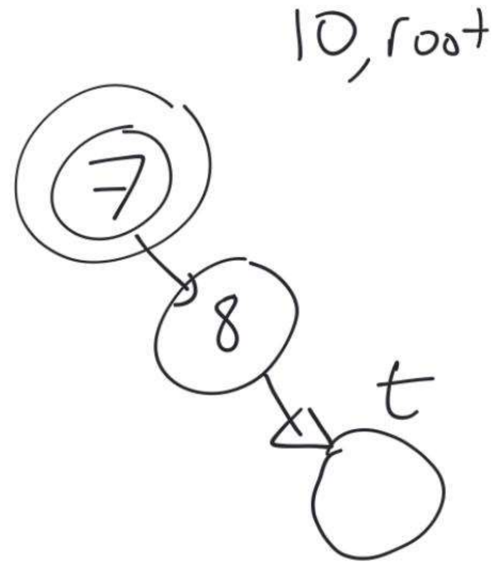
2,3



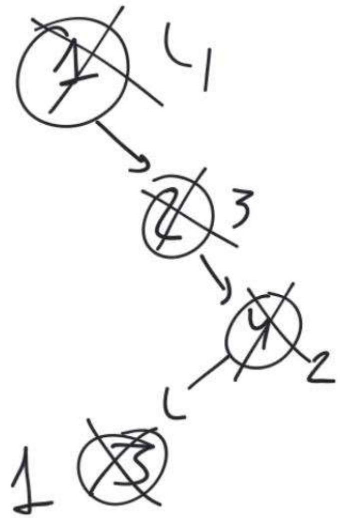
```
// Internal method to remove minimum item from a subtree.
// t is the node that roots the tree.
// Set the new root.
// Throw UnderflowException if t is empty.
template <class Comparable>
void BinarySearchTree<Comparable>::removeMin( Node * & t ) const
{
    if( t == NULL )
        throw UnderflowException( );
    else if( t->left != NULL )
        removeMin( t->left );
    else
    {
        Node *tmp = t;
        t = t->right;
        delete tmp;
    }
}
```



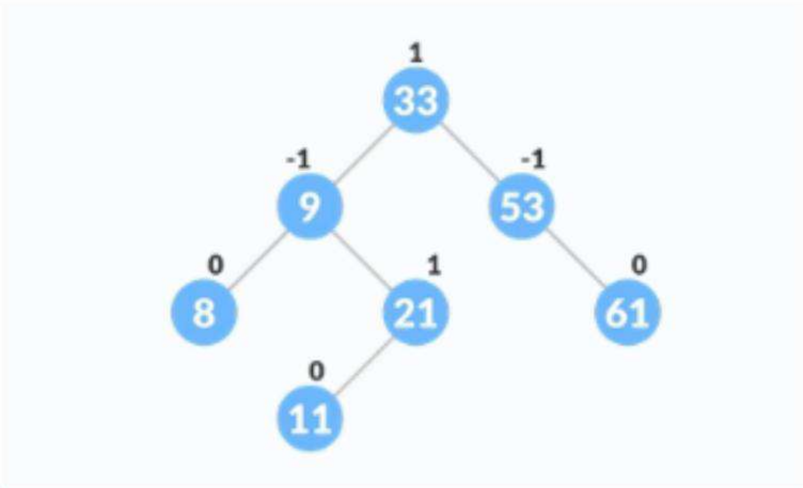
```
// Internal method to find an item in a subtree.
// x is item to search for.
// t is the node that roots the tree.
// Return node containing the matched item.
template <class Comparable>
BinaryNode<Comparable> * BinarySearchTree<Comparable>::
find( const Comparable & x, Node *t ) const
{
    while( t != NULL )
        if( x < t->element )
            t = t->left;
        else if( t->element < x )
            t = t->right;
        else
            return t;    // Match

    return NULL;         // Not found
}
```



```cpp
// Internal method to make subtree empty.
template <class Comparable>
void BinarySearchTree<Comparable>::makeEmpty( Node * & t ) const
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}
```

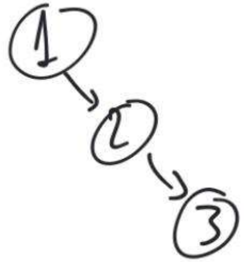


Adelson-Velskii and Landis

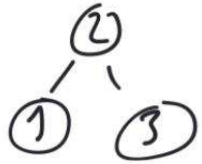


Avl tree

añado algo
evitar
árboles "malos"

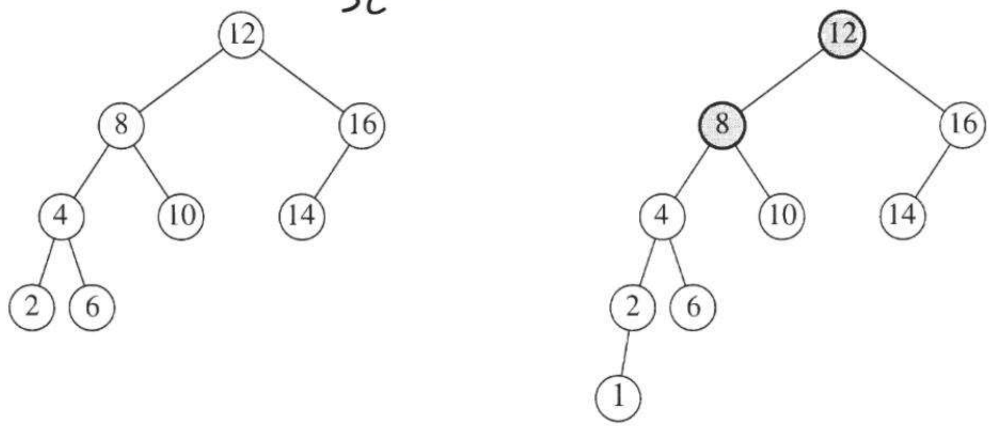


"bueno"



Sí Avl

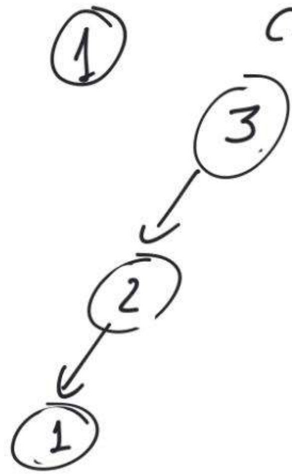
No AVL



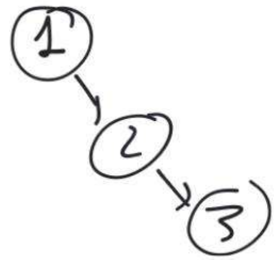
(a) (b)

1. an insertion in the left subtree of the left child of *X*,
2. an insertion in the right subtree of the left child of *X*,
3. an insertion in the left subtree of the right child of *X*, or
4. an insertion in the right subtree of the right child of *X*.

Caso 1



Caso 4



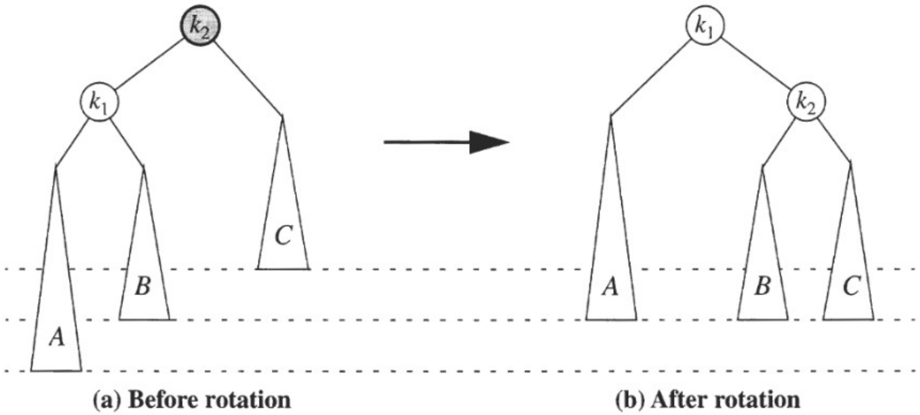


Figure 19.23 Single rotation to fix case 1.

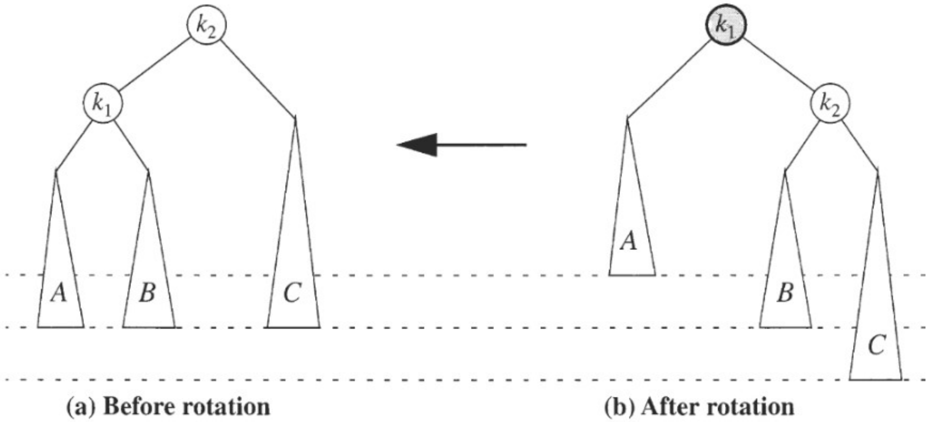


Figure 19.26 Symmetric single rotation to fix case 4.

```
// Rotate binary tree node with left child.
template <class Comparable>
void BST<Comparable>::rotateWithLeftChild( Node * & k2 ) const
{
    Node *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2 = k1;
}
```

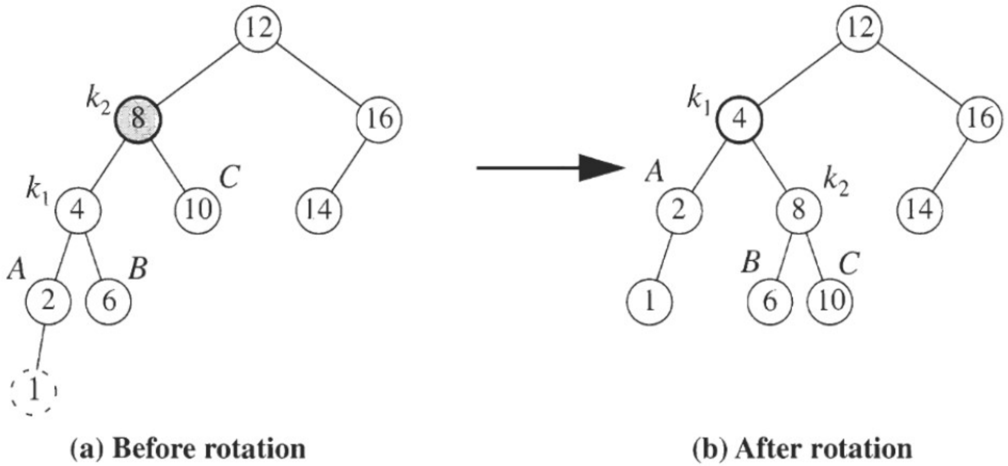


Figure 19.25 Single rotation fixes an AVL tree after insertion of 1.

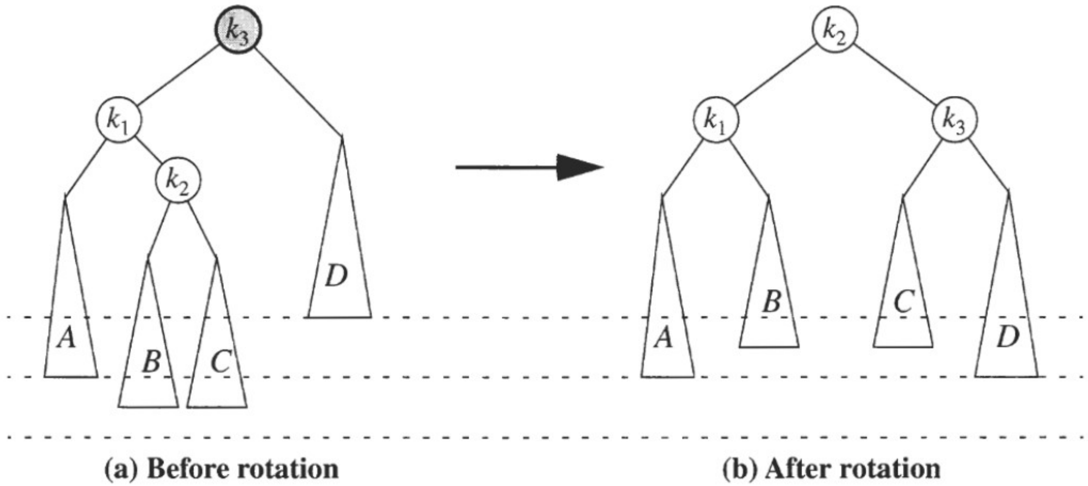


Figure 19.29 Left–right double rotation to fix case 2.

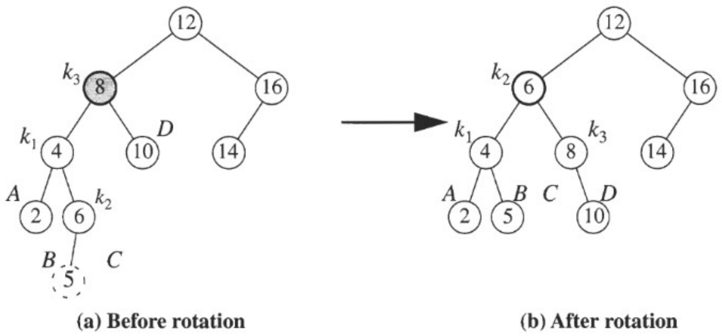
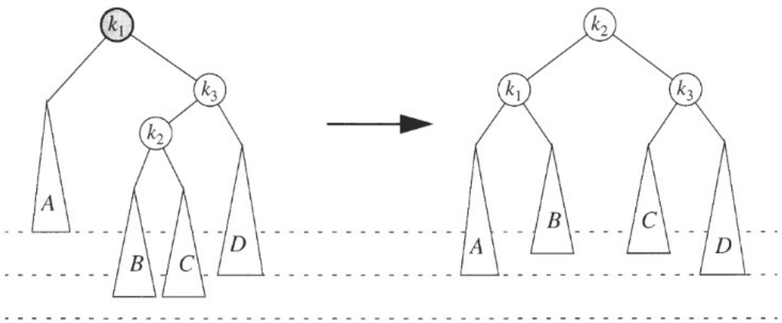


Figure 19.30 Double rotation fixes AVL tree after the insertion of 5.



(a) Before rotation (b) After rotation

Figure 19.31 Left–right double rotation to fix case 3.

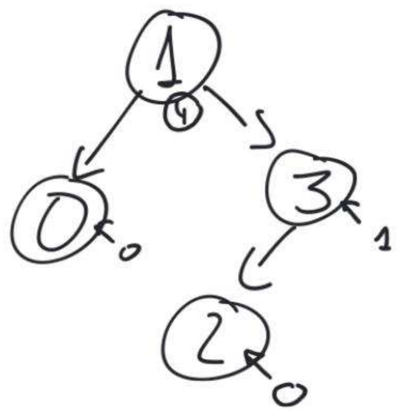
```
// Double rotate binary tree node: first left child
// with its right child; then node k3 with new left child.
// For AVL trees, this is a double rotation for case 2.
template <class Comparable>
void BST<Comparable>::
doubleRotateWithLeftChild( Node * & k3 ) const
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Figure 19.32 Pseudocode for a double rotation (case 2).

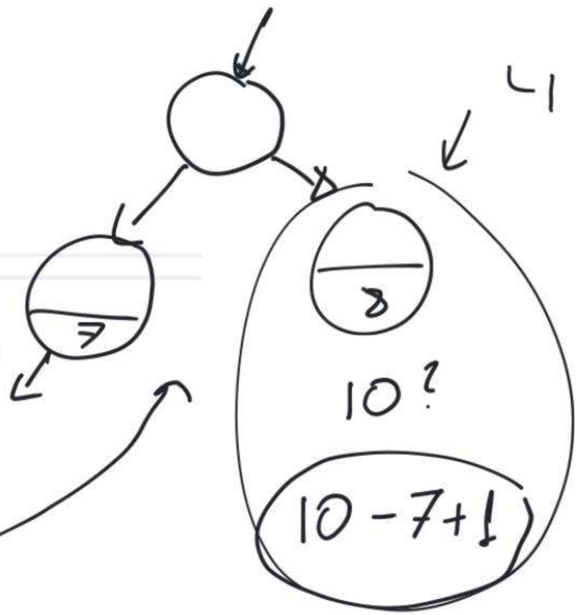
[k]

BST With Rank
Si añadimos una propiedad adicional (size)
Podemos acceder a un determinado elemento rápidamente

```
int treeSize( Node *t ) const
  { return t == NULL ? 0 : t->size; }
```



```cpp
// Returns the kth smallest item in the tree.
// Throws ItemNotFoundException if k is out of range.
template <class Comparable>
Cref<Comparable> BinarySearchTreeWithRank<Comparable>::findKth(int k) const
{
    return elementAt(findKth(k, this->root));
}
// Internal method to find kth item in a subtree.
// k is the desired rank.
// t is the node that roots the tree.
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTreeWithRank<Comparable>::findKth( int k, Node * t ) const
{
    if( t == NULL )
        return NULL;

    int leftSize = treeSize( t->left );

    if( k <= leftSize )
        return findKth( k, t->left );
    else if( k == leftSize + 1 )
        return t;
    else
        return findKth( k - leftSize - 1, t->right );
}
```



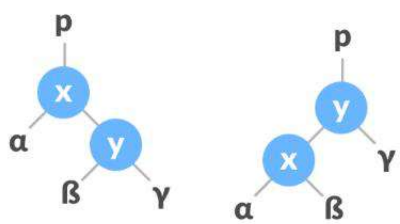
Red-Black tree

A red–black tree is a binary search tree having the following ordering properties:

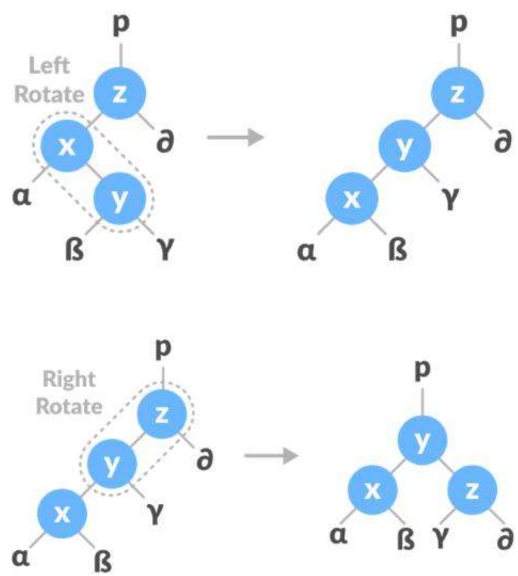
1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a `NULL` pointer must contain the same number of black nodes.

Ejemplo
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

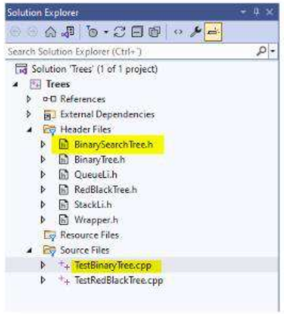
Left Rotate



Left-Right and Right-Left Rotate



Del código que está en BlackBoard
Implementar los métodos que tienen el comentario:
//Implementar



1er trabajar con pair <string,string>
crear un vector de pair<string,string> y almacenar nombres/apellidos
devolver (mediante una función) las personas que tenga igual nombre
y apellido. (el código de ejemplo está en BB)

2o
Implementar el código que falta en el fichero zip que está en el campus
virtual