Arbol rojo y negro

Generated by Doxygen 1.9.3

Chapter 1

Red-black-tree

Red–black tree writen in C++, using templates

1.1 Compiling and execution

To compile the code we use the following command line
```
g++ -o code *.cpp
```

To execute the code on Windows we use the following command line
```
code.exe data.txt
```

To execute the code on Linux we use the following command line
```
./code.out ./data.txt
```

NOTE: you must change [data] by the actual name of the .txt file that you want to study.

1.2 To-do list

- [ ] finish the software design
- [ ] create main.cpp that receive a data .txt file as a parameter
- [ ] create all the .cpp and .hpp/.h files needed for the execution of the code
- [ ] add the internal documentation (Doxygen format)
- [ ] add the external documentation

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

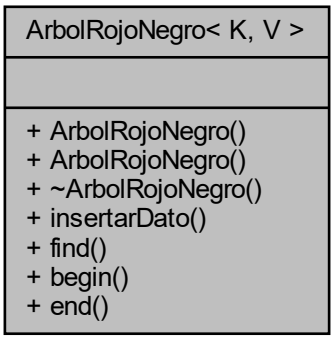
Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 ArbolRojoNegro< K, V > Class Template Reference

Collaboration diagram for ArbolRojoNegro< K, V >:



Classes

- class Iterador

Public Member Functions

- **ArbolRojoNegro** ()

  Constructor por omision de ArbolRojoNegro.
- ArbolRojoNegro (ArbolRojoNegro &&otroArbolMovido)

  Constructor por r-value de ArbolRojoNegro.
- ~**ArbolRojoNegro** ()

  Destructor de ArbolRojoNegro.
- int insertarDato (const V &valor, const K &llave)

Inserta un dato en el arbol.

- Iterador find (const K &llave) const

  El metodo find permite buscar un elemento en el arbol.

- Iterador begin ()

  Retorna un Iterador que apunta a la primera hoja (la de menor valor)

- Iterador end ()

  Retorna un Iterador que apunta al final (nulo)

4.1.1 Constructor & Destructor Documentation

4.1.1.1 ArbolRojoNegro()

```
template<class K , class V >
ArbolRojoNegro< K, V >::ArbolRojoNegro (
            ArbolRojoNegro< K, V > && otroArbolMovido )  [inline]
```

Constructor por r-value de ArbolRojoNegro.

Parameters

otroArbolMovido	r-value de arbol cuyos recursos queremos robar Utilizado para robar recursos de un r-value tras un move

4.1.2 Member Function Documentation

4.1.2.1 begin()

```
template<class K , class V >
Iterador ArbolRojoNegro< K, V >::begin ( )  [inline]
```

Retorna un Iterador que apunta a la primera hoja (la de menor valor)

Returns

Iterador que apunta a la primera hoja

4.1.2.2 end()

```
template<class K , class V >
Iterador ArbolRojoNegro< K, V >::end ( )  [inline]
```

Retorna un Iterador que apunta al final (nulo)

Returns

Iterador nulo

4.1.2.3 find()

```
template<class K , class V >
Iterador ArbolRojoNegro< K, V >::find (
            const K & llave ) const  [inline]
```

El metodo find permite buscar un elemento en el arbol.

Parameters

llave	llave es el unico parametro que recibe el metodo find, es la llave que se tiene que buscar en el arbol para saber si ya existe un elemento en arbol con la misma llave

Returns

El metodo find devuelve un ArbolRojoNegro::Iterador que apunta a la hoja del arbol que contiene el elemento con la llave dada, si no se encontra un elemento con la misma llave entonces signica que no existe en el arbol y por lo tanto devuelve un puntero que apunta a 0 (dirreción hoja nula)

4.1.2.4 insertarDato()

```
template<class K , class V >
int ArbolRojoNegro< K, V >::insertarDato (
            const V & valor,
            const K & llave )  [inline]
```

Inserta un dato en el arbol.

Parameters

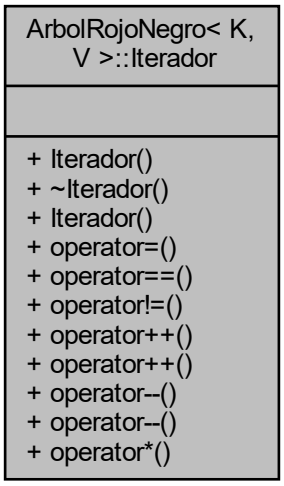
valor	Valor del dato que se inserta
llave	LLave del dato que se inserta

The documentation for this class was generated from the following file:

- ArbolRojoNegro.h

4.2 ArbolRojoNegro< K, V >::Iterador Class Reference

Collaboration diagram for ArbolRojoNegro< K, V >::Iterador:



Public Member Functions

- Iterador (Hoja *hojaIterable)

  Constructor con parametros de Iterador.
- Iterador (const Iterador &iteradorCopiable)

  Constructor por copia de Iterador.
- Iterador & operator= (const Iterador &iteradorCopiable)

  Operador de asignacion para Iterador.
- bool operator== (const Iterador &iteradorComparable)

  Operador "igual que" para Iterador.
- bool operator!= (const Iterador &iteradorComparable)

  Operador "diferente que" para Iterador.
- Iterador & operator++ ()

  Operador de preincremento.
- Iterador operator++ (int x)

  Operador de postincremento.
- Iterador & operator-- ()

  Operador de predecremento.
- Iterador operator-- (int x)

  Operador de postdecremento.
- const V & operator* ()

  Operador de desreferencia.

4.2.1 Constructor & Destructor Documentation

4.2.1.1 Iterador() [1/2]

```
template<class K , class V >
ArbolRojoNegro< K, V >::Iterador::Iterador (
            Hoja * hojaIterable )  [inline]
```

Constructor con parametros de Iterador.

Parameters

hojaIterable	Hoja que se asigna como actual al iterador

4.2.1.2 Iterador() [2/2]

```
template<class K , class V >
ArbolRojoNegro< K, V >::Iterador::Iterador (
            const Iterador & iteradorCopiable )  [inline]
```

Constructor por copia de Iterador.

Parameters

iteradorCopiable	Iterador que se copia

4.2.2 Member Function Documentation

4.2.2.1 operator"!=()

```
template<class K , class V >
bool ArbolRojoNegro< K, V >::Iterador::operator!= (
            const Iterador & iteradorComparable )  [inline]
```

Operador "diferente que" para Iterador.

Parameters

iteradorComparable	Iterador con el que se compara para determinar la desigualdad

Returns

bool que indica si son diferentes

4.2.2.2 operator∗()

```
template<class K , class V >
const V & ArbolRojoNegro< K, V >::Iterador::operator* ( )  [inline]
```

Operador de desreferencia.

Returns

const V& con el valor de la hoja a la que apunta el Iterador

4.2.2.3 operator++() [1/2]

```
template<class K , class V >
Iterador & ArbolRojoNegro< K, V >::Iterador::operator++ ( )  [inline]
```

Operador de preincremento.

Modifica al Iterador para que apunte a la siguiente hoja y lo retorna

Returns

Iterador& que apunta a la siguiente hoja

4.2.2.4 operator++() [2/2]

```
template<class K , class V >
Iterador ArbolRojoNegro< K, V >::Iterador::operator++ (
            int x )  [inline]
```

Operador de postincremento.

Hace una copia del Iterador y modifica al Iterador para que apunte a la siguiente hoja Luego retorna la copia

Parameters

x	Parametro cuya unica funcion es diferenciar el metodo del preincremento

Returns

Iterador& con la copia del Iterador no modificado

4.2.2.5 operator--() [1/2]

```
template<class K , class V >
Iterador & ArbolRojoNegro< K, V >::Iterador::operator-- ( )  [inline]
```

Operador de predecremento.

Modifica al Iterador para que apunte a la hoja anterior y lo retorna

Returns

Iterador& que apunta a la hoja anterior

4.2.2.6 operator--() [2/2]

```
template<class K , class V >
Iterador ArbolRojoNegro< K, V >::Iterador::operator-- (
            int x )  [inline]
```

Operador de postdecremento.

Hace una copia del Iterador y modifica al Iterador para que apunte a la hoja anterior Luego retorna la copia

Parameters

x	Parametro cuya unica funcion es diferenciar el metodo del predecremento

Returns

Iterador& con la copia del Iterador no modificado

4.2.2.7 operator=()

```
template<class K , class V >
Iterador & ArbolRojoNegro< K, V >::Iterador::operator= (
            const Iterador & iteradorCopiable )  [inline]
```

Operador de asignacion para Iterador.

Parameters

iteradorCopiable	Iterador que se asigna

Returns

Iterador& con el Iterador modificado

4.2.2.8 operator==()

```
template<class K , class V >
bool ArbolRojoNegro< K, V >::Iterador::operator== (
            const Iterador & iteradorComparable )  [inline]
```

Operador "igual que" para Iterador.

Parameters

iteradorComparable	Iterador con el que se compara para determinar la igualdad

Returns

bool que indica si son iguales

The documentation for this class was generated from the following file:

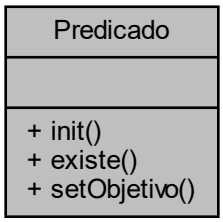
- ArbolRojoNegro.h

4.3 Predicado Class Reference

Clase que contiene un predicado para ser utilizado en find_if.

`#include <Predicado.h>`

Collaboration diagram for Predicado:



Static Public Member Functions

- static void **init** ()

  Inicializa _objetivo.
- static bool existe (const pair< string, string > &llaveValor)

  Determina si la palabra se encuentra como primer elemento en el par.
- static void setObjetivo (string)

  Metodo set de objetivo.

4.3.1 Detailed Description

Clase que contiene un predicado para ser utilizado en find_if.

4.3.2 Member Function Documentation

4.3.2.1 existe()

```
bool Predicado::existe (
            const pair< string, string > & parejaValor )  [static]
```

Determina si la palabra se encuentra como primer elemento en el par.

Parameters

parejaValor	Pair que contien dos strings

Returns

bool que indica si se encuentra

4.3.2.2 setObjetivo()

```
void Predicado::setObjetivo (
            string objetivo )  [static]
```

Metodo set de objetivo.

Parameters

objetivo	String que se le asigna a objetivo

The documentation for this class was generated from the following files:

- Predicado.h
- Predicado.cpp

Chapter 5

File Documentation

5.1 ArbolRojoNegro.h

```
1 #ifndef _ARBOLROJONEGRO_
2 #define _ARBOLROJONEGRO_
3
4 #include <iostream> // Para imprimir los mensajes de debugging
5
6 template <class K, class V> //K es llave, V es valor
7
8 class ArbolRojoNegro
9 {
10     private:
11
12         class Connector
13         {
14             public:
15
16                 K llave;
17                 char tipo;      //differenciar si es nodo o hoja --> nodo = 0, hoja = 1
18
19                 static const char negro = 1;
20                 static const char rojo = 0;
21
22                 static const char tipoNodo = 1;
23                 static const char tipoHoja = 0;
24
25                 Connector(){}; // Tiene cuerpo porque sub-clases llaman a los constructores en jerarquía,
      incluso si esta clase es abstracta
26                 virtual ~Connector(){}; // También tiene cuerpo porque sub-clases llaman a los
      destructores en jerarquía, incluso si esta clase es abstracta
27         };
28
29         class Hoja : public Connector
30         {
31             public:
32
33                 V valor;
34                 Hoja* next;
35                 Hoja* previous;
44                 Hoja(const V& valorTemporal, const K& llaveTemporal)
45                 {
46                     this->valor = valorTemporal;
47                     this->llave = llaveTemporal;
48
49                     this->next = 0;
50                     this->previous = 0;
51
52                     this->tipo = ArbolRojoNegro::Connector::tipoHoja;
53                 }
54                 ~Hoja(){};
55         };
56
57         class Nodo : public Connector
58         {
59             public:
60                 static const char ladoIzquierdo = 0;
61                 static const char ladoDerecho = 1;
62
63                 Connector* hijos[2];   //0 es izquierda, 1 es derecha
64                 char color;
```

```
65
66                 Nodo();
67                 ~Nodo()
68                 {if(hijos[0] != 0) delete hijos[0]; if(hijos[1] != 0) delete hijos[1];}
69
77                 Nodo(Hoja* hoja_A, Hoja* hoja_B, char color)
78                 {
79                     this->tipo = ArbolRojoNegro::Connector::tipoNodo;
80                     this->color = color;
81
82                     if(hoja_A == 0 || hoja_B == 0) this->llave = K();
83                     else if(hoja_A->llave < hoja_B->llave)
84                     {
85                         this->llave = hoja_A->llave;
86
87                         this->hijos[ladoIzquierdo] = hoja_A;
88                         this->hijos[ladoDerecho] = hoja_B;
89
90                         // Caso 1 -> La hoja A es foránea a la lista doblemente enlazada
91                         if(hoja_A->previous == 0 && hoja_A->next == 0)
92                         {
93                             hoja_A->previous = hoja_B->previous;
94                             if(hoja_B->previous != 0) hoja_B->previous->next = hoja_A;
95                         }
96                         // Caso 2 -> La hoja A es nativa a la lista doblemente enlazada
97                         else
98                         {
99                             hoja_B->next = hoja_A->next;
100                             if(hoja_A->next != 0) hoja_A->next->previous = hoja_B;
101                         }
102
103                         hoja_A->next = hoja_B;
104                         hoja_B->previous = hoja_A;
105                     }
106                     else
107                     {
108                         this->llave = hoja_B->llave;
109
110                         this->hijos[ladoDerecho] = hoja_A;
111                         this->hijos[ladoIzquierdo] = hoja_B;
112
113                         // Caso 1 -> La hoja B es foránea a la lista doblemente enlazada
114                         if(hoja_B->previous == 0 && hoja_B->next == 0)
115                         {
116                             hoja_B->previous = hoja_A->previous;
117                             if(hoja_A->previous != 0) hoja_A->previous->next = hoja_B;
118                         }
119                         // Caso 2 -> La hoja B es nativa a la lista doblemente enlazada
120                         else
121                         {
122                             hoja_A->next = hoja_B->next;
123                             if(hoja_B->next != 0) hoja_B->next->previous = hoja_A;
124                         }
125
126                         hoja_B->next = hoja_A;
127                         hoja_A->previous = hoja_B;
128                     }
129                 }
133                 void colorFlipLocal()
134                 {
135                     if(this->color == (char)0) this->color = (char)1;
136                     else this->color = (char)0;
137
138                     return;
139                 }
140         };
141
142         Connector* raiz; //pila -> mejor O
143         Hoja* hojaMinima;
144
149         void CF(Nodo* padre)
150         {
151             if(padre == 0) return; // Código defensivo
152
153             // Casting se asume seguro porque padre debe ser un nodo
154             if(padre != raiz) padre -> colorFlipLocal();
155
156             // Se asume que el árbol minimal de nodos existe, y ningulo de los dos hijos es nulo u hoja
157             dynamic_cast<Nodo*>(padre->hijos[0]) -> colorFlipLocal();
158             dynamic_cast<Nodo*>(padre->hijos[1]) -> colorFlipLocal();
159         }
164         void RSI(Connector** bis)
165         {
166             if(bis == 0) return; // Código defensivo
167             Nodo* nodoBis = dynamic_cast<Nodo*>(*bis);
168
169             // Casting se asume seguro porque bisabuelo debe apuntar a un nodo
```

```
170             Nodo* nuevoPadre = dynamic_cast<Nodo *>(nodoBis->hijos[Nodo::ladoDerecho]);
171 
172             // Se asume que la cadena de nodos existe, y ningulo es nulo
173             nodoBis->hijos[Nodo::ladoDerecho] = nuevoPadre->hijos[Nodo::ladoIzquierdo];
174             nuevoPadre->hijos[0] = (* bis);
175 
176             (* bis) = nuevoPadre; // std::cout « "RSI EXITOSA" « std::endl;
177             RC(nuevoPadre);
178 
179             return;
180         }
185         void RSD(Connector** bis)
186         {
187             if(bis == 0) return; // Código defensivo
188             Nodo* nodoBis = dynamic_cast<Nodo*>(*bis);
189 
190             // Casting se asume seguro porque bisabuelo debe apuntar a un nodo
191             Nodo* nuevoPadre = dynamic_cast<Nodo *>(nodoBis->hijos[0]);
192 
193             // Se asume que la cadena de nodos existe, y ningulo es nulo
194             nodoBis->hijos[0] = nuevoPadre->hijos[1];
195             nuevoPadre->hijos[1] = (* bis);
196 
197             (* bis) = nuevoPadre;
198             RC(nuevoPadre);
199 
200             return;
201         }
206         void RDI(Connector** bis)
207         {
208             if(bis == 0) return; // Código defensivo
209             Nodo* nodoBis = dynamic_cast<Nodo*>(*bis);
210 
211             Nodo* antiguoPadre = dynamic_cast<Nodo * >(nodoBis->hijos[1]);
212             Nodo* nuevoPadre = dynamic_cast<Nodo * >(antiguoPadre->hijos[0]);
213 
214             nodoBis->hijos[1] = nuevoPadre->hijos[0];
215             antiguoPadre->hijos[0] = nuevoPadre->hijos[1];
216             nuevoPadre->hijos[0] = (* bis);
217             nuevoPadre->hijos[1] = antiguoPadre;
218 
219             (*bis) = nuevoPadre;
220             RC(nuevoPadre);
221 
222             return;
223         }
228         void RDD(Connector** bis)
229         {
230             if(bis == 0) return; // Código defensivo
231             Nodo* nodoBis = dynamic_cast<Nodo*>(*bis);
232 
233             Nodo * antiguoPadre = dynamic_cast<Nodo*>(nodoBis->hijos[0]);
234             Nodo * nuevoPadre = dynamic_cast<Nodo*>(antiguoPadre->hijos[1]);
235 
236             nodoBis->hijos[0] = nuevoPadre->hijos[1];
237             antiguoPadre->hijos[1] = nuevoPadre->hijos[0];
238             nuevoPadre->hijos[1] = (*bis);
239             nuevoPadre->hijos[0] = antiguoPadre;
240 
241             (*bis) = nuevoPadre;
242             RC(nuevoPadre);
243 
244             return;
245         }
246 
251         void RC(Nodo* padre)
252         {
253             // std::cout « "ENTRANDO A RC" « std::endl;
254 
255             if(padre == 0) return; // Código defensivo
256             padre -> color = Nodo::negro;
257 
258             // Se asume que los castings son seguros porque se asume que ambos hijos existen, y son
    nodos
259             dynamic_cast<Nodo*>(padre->hijos[Nodo::ladoIzquierdo]) -> color = Nodo::rojo;
260             dynamic_cast<Nodo*>(padre->hijos[Nodo::ladoDerecho]) -> color = Nodo::rojo;
261 
262             // std::cout « "RC EXITOSO" « std::endl;
263         }
264 
270         char verificarRotacion(Connector** bisabuelo)
271         {
272             Nodo* padre = dynamic_cast<Nodo*>(*bisabuelo);
273 
274             if(padre->hijos[0] -> tipo == ArbolRojoNegro::Connector::tipoNodo)
275             {
276                 Nodo* nodoHijoIzquierdo = dynamic_cast<Nodo*>(padre->hijos[0]);
```

```
277
278                 if(nodoHijoIzquierdo-> color == ArbolRojoNegro::Connector::rojo) // Nodo hijo izquierdo
    es rojo
279                 {
280                     if(nodoHijoIzquierdo -> hijos[0] -> tipo == ArbolRojoNegro::Connector::tipoNodo)
281                     {
282                         Nodo* nietoHijoIzquierdo = dynamic_cast<Nodo*>(nodoHijoIzquierdo->hijos[0]);
283
284                         if(nietoHijoIzquierdo-> color == ArbolRojoNegro::Connector::rojo) return
    char(2);   // hijo izquierdo del hijo izquierdo es rojo
285                         // rotacion simple derecha
286                     }
287
288                     else if(nodoHijoIzquierdo -> hijos[1] -> tipo ==
    ArbolRojoNegro::Connector::tipoNodo)
289                     {
290                         Nodo* nietoHijoDerecho = dynamic_cast<Nodo*>(nodoHijoIzquierdo->hijos[1]);
291
292                         if(nietoHijoDerecho-> color == ArbolRojoNegro::Connector::rojo) return char(4);
    // hijo derecho del hijo izquierdo es rojo
293                         // rotacion doble derecha
294                     }
295                 }
296             }
297
298             else if(padre->hijos[1] -> tipo == ArbolRojoNegro::Connector::tipoNodo)
299             {
300                 Nodo* nodoHijoDerecho = dynamic_cast<Nodo*>(padre->hijos[1]);
301
302                 if(nodoHijoDerecho-> color == ArbolRojoNegro::Connector::rojo) // Nodo hijo derecho es
    rojo
303                 {
304                     if(nodoHijoDerecho -> hijos[0] -> tipo == ArbolRojoNegro::Connector::tipoNodo)
305                     {
306                         Nodo* nietoHijoIzquierdo = dynamic_cast<Nodo*>(nodoHijoDerecho->hijos[0]);
307
308                         if(nietoHijoIzquierdo-> color == ArbolRojoNegro::Connector::rojo) return
    char(3);   // hijo izquierdo del hijo derecho es rojo
309                         // rotacion doble izquierda
310                     }
311
312                     else if(nodoHijoDerecho -> hijos[1] -> tipo == ArbolRojoNegro::Connector::tipoNodo)
313                     {
314                         Nodo* nietoHijoDerecho = dynamic_cast<Nodo*>(nodoHijoDerecho->hijos[1]);
315
316                         if(nietoHijoDerecho-> color == ArbolRojoNegro::Connector::rojo) return char(1);
    // hijo derecho del hijo derecho es rojo
317                         // rotacion simple izquierda
318                     }
319
320                     else return char(1); // hijo izquierdo es rojo pero nieto no lo es
321                 }
322             }
323
324             // Si ninguna rotación se debe realizar (el criterio de ninguna se cumple completamente)
    entonces devolvemos 0
325             return char(0);
326         }
327
328      public:
329
330         class Iterador
331         {
332             private:
333                 Hoja* actual;
334
335             public:
341                 Iterador(Hoja* hojaIterable)
342                 {this->actual = hojaIterable;}
343                 ~Iterador(){};
348                 Iterador(const Iterador& iteradorCopiable)
349                 {this->actual = iteradorCopiable.actual;}
350
351
358                 Iterador& operator=(const Iterador& iteradorCopiable)
359                 {
360                     this->actual = iteradorCopiable.actual;
361                     return *this;
362                 }
363
370                 bool operator==(const Iterador& iteradorComparable)
371                 {return (this->actual == iteradorComparable.actual);}
378                 bool operator!=(const Iterador& iteradorComparable)
379                 {return (this->actual != iteradorComparable.actual);}
380
387                 Iterador& operator++()
388                 {
```

```
389                     actual = actual->next;
390                     return *this;
391                 }
400                 Iterador operator++(int x)
401                 {
402                     Iterador copia(*this);
403                     actual = actual->next;
404                     return copia;
405                 }
412                 Iterador& operator--()
413                 {
414                     actual = actual->previous;
415                     return *this;
416                 }
425                 Iterador operator--(int x)
426                 {
427                     Iterador copia(*this);
428                     actual = actual->previous;
429                     return copia;
430                 }
431
437                 const V& operator*()
438                 {return this->actual->valor;}
439         };
440
444         ArbolRojoNegro()
445         {
446             raiz = 0;
447             hojaMinima = 0;
448         }
449
455         ArbolRojoNegro(ArbolRojoNegro&& otroArbolMovido)
456         {
457             raiz = otroArbolMovido.raiz;
458             hojaMinima = otroArbolMovido.hojaMinima;
459
460             otroArbolMovido.raiz = 0;
461         }
462
466         ~ArbolRojoNegro()
467         {
468             // Desctructor recursivo, en cadena
469             if(raiz != 0) delete raiz;
470         }
471
478         int insertarDato(const V& valor, const K& llave)
479         {
480             // Recorrer el arbol, hacer el color flip si se occupa, sino, no
481             // InsertarHoja
482             // Verificar que sea un arbol RN
483             // Si no lo es entonces hacer rotacion y arreglar colores
484
485             // Caso trivial 1 -> Arbol vacio
486             if(raiz == 0)
487             {
488                 Hoja* nuevaHoja = new Hoja(valor, llave);
489
490                 raiz = (Connector*) nuevaHoja;
491                 hojaMinima = nuevaHoja;
492
493                 return 1;
494             }
495
496             // Caso trivial 2 -> Llave ya existe en la raiz. Comparacion es segura porque raiz no es
    nula (caso trivial 1)
497             if(raiz->llave == llave) return 0;
498
499             // Caso trivial 3 -> Arbol solo tiene 1 hoja.
500             // Comparacion es necesaria porque raiz tiene llave distinta (caso trivial 2), y es segura
    porque raiz no es nula (caso trivial 1)
501             if(raiz->tipo == ArbolRojoNegro::Connector::tipoHoja)
502             {
503                 Hoja* nuevaHoja = new Hoja(valor, llave);
504                 Hoja* raizComoHoja = dynamic_cast<Hoja*>(raiz); // Casting es seguro porque raiz era una
    Hoja
505                 Nodo* nuevoNodo;
506
507                 // En ambos casos el nuevo nodo raíz será negro de una vez
508                 nuevoNodo = new Nodo(nuevaHoja, raizComoHoja, ArbolRojoNegro::Connector::negro);
509
510                 if(llave < raiz->llave) hojaMinima = nuevaHoja;
511                 else hojaMinima = raizComoHoja;
512
513                 raiz = (Connector*) nuevoNodo;
514                 return 1;
515             }
516
```

```
517             // Casting es seguro, confirmamos que raiz no es nula (caso trivial 1) y es nodo (caso
    trivial 3)
518             Nodo* nodoActual = dynamic_cast<Nodo*>(raiz);
519             char ladoActual = 0;
520 
521             // Conocemos si debemos mover a la izquierda o derecha. La comparación es segura porque
    porque raíz no es nula (caso trivial 1)
522             // Confirmamos que si no es mayor entonces es menor. No puede ser igual (caso trivial 2)
523             if(llave < raiz->llave) ladoActual = Nodo::ladoIzquierdo;
524             else ladoActual = Nodo::ladoDerecho;
525 
526             // Por estructura del árbol, se garantiza que los hijos de un nodo jamás serán nulos
527             // Quizás deberíamos poner código defensivo aquí, por si acaso
528             Connector* connectorHijoActual = nodoActual->hijos[ladoActual];
529 
530             // Caso semi-trivial -> Connector siguiente es Hoja
531             if(connectorHijoActual->tipo == ArbolRojoNegro::Connector::tipoHoja)
532             {
533                 // Subcaso trivial del caso semi-trivial -> Llave ya existe.
534                 // Comparacion se asume segura porque el hijo connector se asume que no es nulo
535                 if(connectorHijoActual->llave == llave) return 0;
536 
537                 Hoja* nuevaHoja = new Hoja(valor, llave);
538                 Hoja* hojaHijaActual = dynamic_cast<Hoja*>(connectorHijoActual);
539                 Nodo* nuevoNodo = new Nodo(hojaHijaActual, nuevaHoja, ArbolRojoNegro::Connector::rojo);
540 
541                 nodoActual->hijos[ladoActual] = (Connector*) nuevoNodo;
542                 if(llave < hojaMinima->llave) hojaMinima = nuevaHoja;
543 
544                 return 1;
545             }
546 
547             // Caso de solución iterativa -> Connector siguiente es un nodo
548             // Casting cada ciclo es seguro, porque se garantiza que el conector siguiente es un nodo
549 
550             // Vamos a descender en el árbol hasta el hijo correspondiente al nodo actual sea una hoja
551             unsigned char bandera = 0; // Existe un desfase de dos entre el nodo actual y el bisabuelo,
    no todas las iteraciones determinan el bisabuelo
552             Connector** bis = 0; // Ubicación del bisasbuelo
553             Connector** bis_desfasado = 0; // Bis desfasado 1 paso
554             char lado_desfasado_1 = 0; // Lado desfasado 1 paso
555 
556             while(connectorHijoActual->tipo == ArbolRojoNegro::Connector::tipoNodo)
557             {
558                 // std::cout « "Es hora de descender!" « std::endl;
559                 // Antes de descender, verificaremos si ambos hermanos son rojos, si acaso es necesario
    realizar un color flip en el nodo actual
560                 Connector* hermanoTemporal;
561                 if(ladoActual == Nodo::ladoIzquierdo) hermanoTemporal =
    nodoActual->hijos[Nodo::ladoDerecho];
562                 else hermanoTemporal = nodoActual->hijos[Nodo::ladoIzquierdo];
563 
564                 if(hermanoTemporal->tipo == ArbolRojoNegro::Connector::tipoNodo) // Comparación es
    segura porque la estructura del árbol evita connectores nulos
565                 {
566                     // std::cout « "HAY NODO HERMANO" « std::endl;
567                     // Castings son seguros porque se garantizó que ambos conectores existen y son nodos
568                     Nodo* nodoHermano = dynamic_cast<Nodo*>(hermanoTemporal);
569                     Nodo* nodoHijo = dynamic_cast<Nodo*>(connectorHijoActual);
570 
571                     if(nodoHermano->color == Nodo::rojo && nodoHijo->color == Nodo::rojo)
572                     {
573                         // std::cout « "Es hora de hacer color flip!" « std::endl;
574                         CF(nodoActual);
575                     }
576                 }
577 
578                 // Ahora sí, podemos descender.
579                 if(bandera == 0) // El primer descenso es especial, está garantizado, y se conoce el
    bisabuelo, pues es la dirección de la raíz
580                 {
581                     // std::cout « "-> bandera es 0" « std::endl;
582                     bis = &raiz;
583                     lado_desfasado_1 = ladoActual;
584 
585                     bandera += 1;
586                 }
587                 else // El resto de descensos son fácilmente iterables
588                 {
589                     // std::cout « "-> bandera es > 0" « std::endl;
590                     bis_desfasado = bis;
591                     bis = &(dynamic_cast<Nodo*>(*bis_desfasado)->hijos[lado_desfasado_1]);
592 
593                     lado_desfasado_1 = ladoActual;
594                 }
595 
596                 nodoActual = dynamic_cast<Nodo*>(connectorHijoActual);
```

```
597
598                 // std::cout « "-> raiz apunta a " « (*raiz).llave « std::endl;
599                 // std::cout « "-> bis apunta a " « (*bis)->llave « std::endl;
600                 // std::cout « "-> nodo actual es " « nodoActual->llave « std::endl;
601                 // std::cout « "-> hijo izq es " « nodoActual->hijos[Nodo::ladoIzquierdo]->llave «
    std::endl;
602                 // std::cout « "-> hijo der es " « nodoActual->hijos[Nodo::ladoDerecho]->llave «
    std::endl;
603
604                 if(llave == nodoActual->llave) return 0; // Llave preexistente, no vale la pena seguir
    bajando
605
606                 if(llave < nodoActual->llave) ladoActual = Nodo::ladoIzquierdo;
607                 else ladoActual = Nodo::ladoDerecho;
608
609                 connectorHijoActual = nodoActual->hijos[ladoActual];
610
611                 // std::cout « "-> hijo actual ahora es " « connectorHijoActual->llave « std::endl;
612                 // std::cout « "-> hijo actual ahora es ";
613                 // if(connectorHijoActual->tipo == ArbolRojoNegro::Connector::tipoHoja) std::cout « "UNA
    HOJA" « std::endl;
614                 // else std::cout « "UN NODO" « std::endl;
615             }
616
617             // std::cout « "Es hora de insertar!" « std::endl;
618
619             // Tenemos que el connector siguiente es una hoja, y podemos realizar una inserción
620             if(llave == connectorHijoActual->llave) return 0; // Llave ya preexistente
621
622             Hoja* hojaHijaActual = dynamic_cast<Hoja*>(connectorHijoActual);
623             Hoja* nuevaHoja = new Hoja(valor, llave);
624             Nodo* nuevoNodo = new Nodo(nuevaHoja, hojaHijaActual, ArbolRojoNegro::Connector::rojo);
625
626             nodoActual->hijos[ladoActual] = (Connector*) nuevoNodo;
627             if(llave < hojaMinima->llave) hojaMinima = nuevaHoja;
628
629             // std::cout « "Es hora de verificar rotaciones!" « std::endl;
630
631             // Realizamos la rotación necesaria, si acaso
632             char codigoRotacion = verificarRotacion(bis);
633             // std::cout « "Rotaciones verificadas!" « std::endl;
634
635             switch (codigoRotacion)
636             {
637                 case char(0):
638                     // std::cout « "NO HAY QUE HACER NADA" « std::endl;
639                     break;
640
641                 case char(1):
642                     // std::cout « "-> rotacion simple izquierda" « std::endl;
643                     RSI(bis);
644                     break;
645
646                 case char(2):
647                     // std::cout « "-> rotacion simple derecha" « std::endl;
648                     RSD(bis);
649                     break;
650
651                 case char(3):
652                     // std::cout « "-> rotacion doble izquierda" « std::endl;
653                     RDI(bis);
654                     break;
655
656                 case char(4):
657                     // std::cout « "-> rotacion doble derecha" « std::endl;
658                     RDD(bis);
659                     break;
660
661                 default:
662                     // std::cout « "CODIGO INVALIDO" « std::endl;
663                     break;
664             }
665
666             return 1;
667         }
673         Iterador find(const K& llave) const
674         {
675             if(raiz == 0) return Iterador(0); // Caso trivial, la raiz es nula, no se puede buscar
676
677             Connector* actual = raiz;
678             while(actual->tipo == ArbolRojoNegro::Connector::tipoNodo)
679             {
680                 Nodo* nodoActual = dynamic_cast<Nodo*>(actual); // Este casting es seguro porque este
    connector es un nodo
681                 char ladoActual = 0;
682
683                 if(actual->llave < llave) ladoActual = Nodo::ladoDerecho;
```

```
684                 else ladoActual = Nodo::ladoIzquierdo;
685
686                 // std::cout « "Llave recibida es " « llave « std::endl;
687                 // std::cout « "Llave de connector actual es " « actual->llave « std::endl;
688
689                 // std::cout « "Color de conector actual es ";
690                 // if(nodoActual->color == Nodo::negro) std::cout « "NEGRO" « std::endl;
691                 // else std::cout « "ROJO" « std::endl;
692
693                 // std::cout « "Lado que vamos a tomar es ";
694                 // if(ladoActual == Nodo::ladoIzquierdo) std::cout « "IZQUIERDO" « std::endl;
695                 // else std::cout « "DERECHO" « std::endl;
696
697                 // El nuevo conector actual va a ser el hijo de este conector actual, según el lado
    elegido mediante la llave
698                 actual = nodoActual->hijos[ladoActual];
699             }
700
701             // Si llegamos a la hoja y la llave coincide, encontramos el valor
702             if(actual->llave == llave) return Iterador(dynamic_cast<Hoja *>(actual)); // Este casting es
    seguro porque este connector es una hoja
703             else return Iterador(0); // Si no, retornamos un iterador a dirección de hoja nula
704 }
710         Iterador begin()
711         {return Iterador(hojaMinima);}
717         Iterador end()
718         {return Iterador(0);}
719 };
720
721 #endif
```

5.2 Predicado.h

```
1 #ifndef _PREDICADO
2 #define _PREDICADO
3 #include <string>
4 #include <vector>
5 #include <utility>
6
7 using namespace std;
8
12 class Predicado{
13     private:
14         inline static string _objetivo;
15     public:
16         static void init();
17         static bool existe(const pair<string,string>& llaveValor);
18         static void setObjetivo(string);
19 };
20
21 #endif
```

Index