# Universidad de Buenos Aires
# Facultad de Ingeniería

# 75.06 Organización de Datos
# Cátedra Servetto

# Trabajo Practico I

**Grupo Gordon Moore**
**Integrantes:**
- 80784 Rodrigo Bengoechea
- 74332 Francisco Dibar

# Introducción

SVN es un sistema controlador de versiones, es decir, administra archivos y directorios, y los cambios efectuados sobre los mismos a través del tiempo. Esto permite que un programador pueda recuperar versiones anteriores de su proyecto, o examinar el historial de cambios sobre el mismo.

## Componentes

SVN esta compuesto de 2 componentes (ejecutables): svnadmin y svnuser.
El primero sirve para la administración del almacén de repositorios (creación, eliminación, etc.), de los repositorios en si y de los usuarios.
El segundo es utilizado por los usuarios para almacenar y recuperar su trabajo.

## Instalación y Compilación

- Descomprimir el archivo svngrupo.tar.gz en un directorio destino
- Ejecutar make

Dependencias:
- Xerces (libxerces27)
Es necesario tener esta librería instalada para poder parsear y generar los archivos xml.

# Descripción de los comandos

## svnadmin

**Como crear el almacén de repositorios:**
./svnadmin -a "nombre del almacen"
Este comando genera un directorio llamado "nombre del almacen" donde se almacenaran los archivos correspondientes al mismo.
A su vez, se genera un archivo llamado ~/.svn_grupo_config donde se guarda la configuración del mismo en formato xml.

**Como crear un repositorio:**
./svnadmin -c "nombre del repositorio"
Este comando genera un directorio llamado "nombre del repositorio" dentro del directorio del almacén, donde se almacenaran los archivos correspondientes al mismo.

**Como eliminar un repositorio:**
./svnadmin -r "nombre del repositorio"
Este comando además elimina todos los usuarios pertenecientes al repositorio.

**Como agregar un usuario:**
./svnadmin -u usuario password "nombre del usuario" "nombre del repositorio"

**Como eliminar un usuario:**
./svnadmin -e usuario "nombre del repositorio"

**Como obtener ayuda:**
./svnadmin -h

**Como obtener el listado de usuarios:**
./svnadmin -o "nombre del repositorio"

**Como obtener el listado de los últimos cambios efectuados por un usuario:**
./svnadmin -m "nombre del repositorio" [["nombre del usuario"] [nro de cambios]]

# svnuser

## Como almacenar archivos y directorios:

./svnuser usuario contraseña -a "nombre del repositorio" "nombre del archivo o directorio"
Los archivos se identifican de la misma manera en que fueron ingresados, o sea, el archivo ~/f1 es distinto a /home/user/f1

## Como ver diferencias entre dos versiones del repositorio:

./svnuser usuario contraseña -d "nombre del repositorio" version_inicial version_final

## Como ver las actualizaciones en una fecha dada:

./svnuser usuario contraseña -f "nombre del repositorio" fecha (aaaa/mm/dd)

## Como obtener ayuda:

./svnuser -h

## Como ver el historial de cambios a un archivo o directorio:

./svnuser usuario contraseña -l "nombre del repositorio" "nombre del archivo o directorio"

## Como ver el listado de los últimos cambios efectuados por un usuario:

./svnuser usuario contraseña -m "nombre del repositorio" [cantidad]
donde 'cantidad' indica los últimos 'cantidad' de cambios.

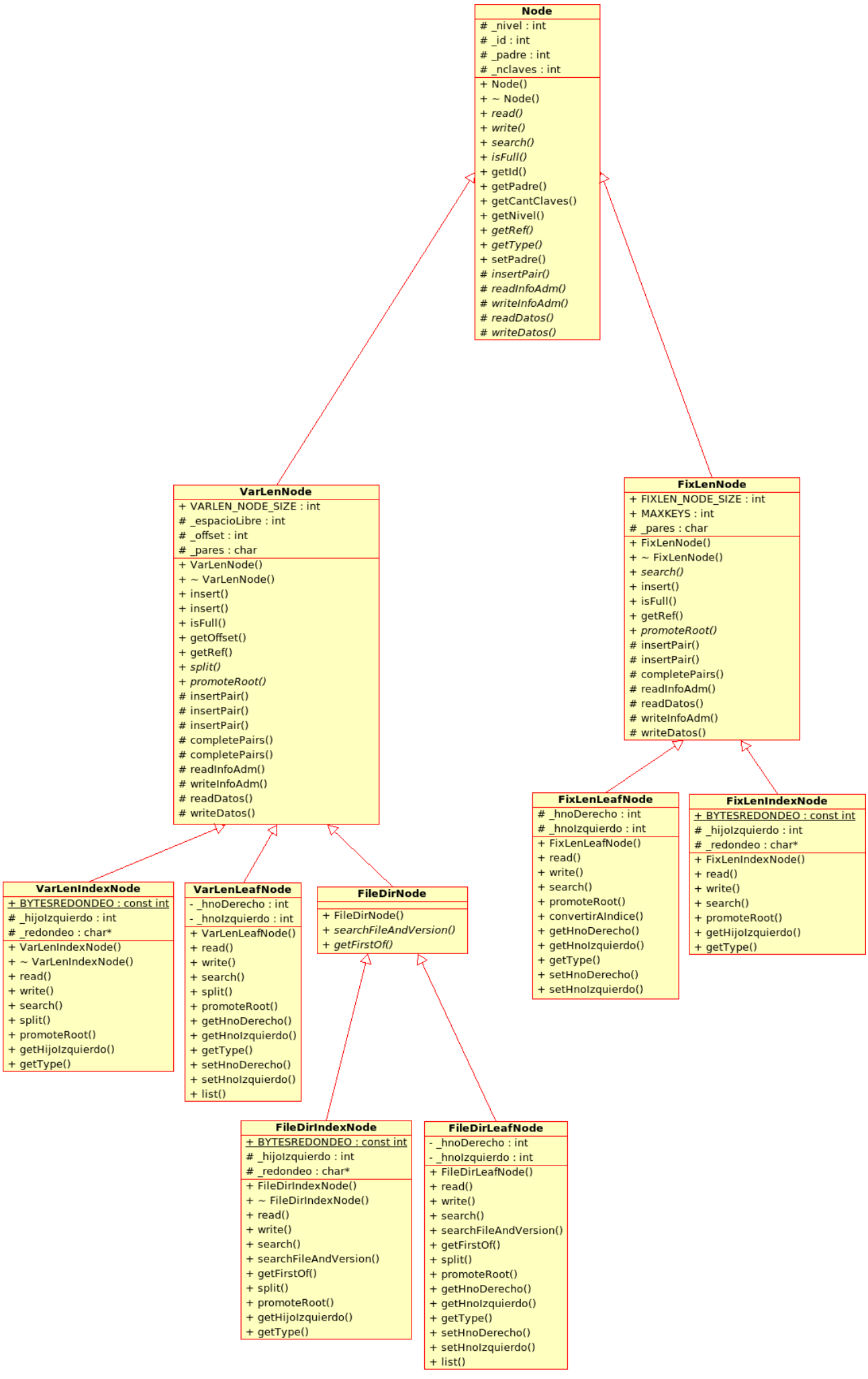
## Como obtener una determinada version de un archivo o directorio:

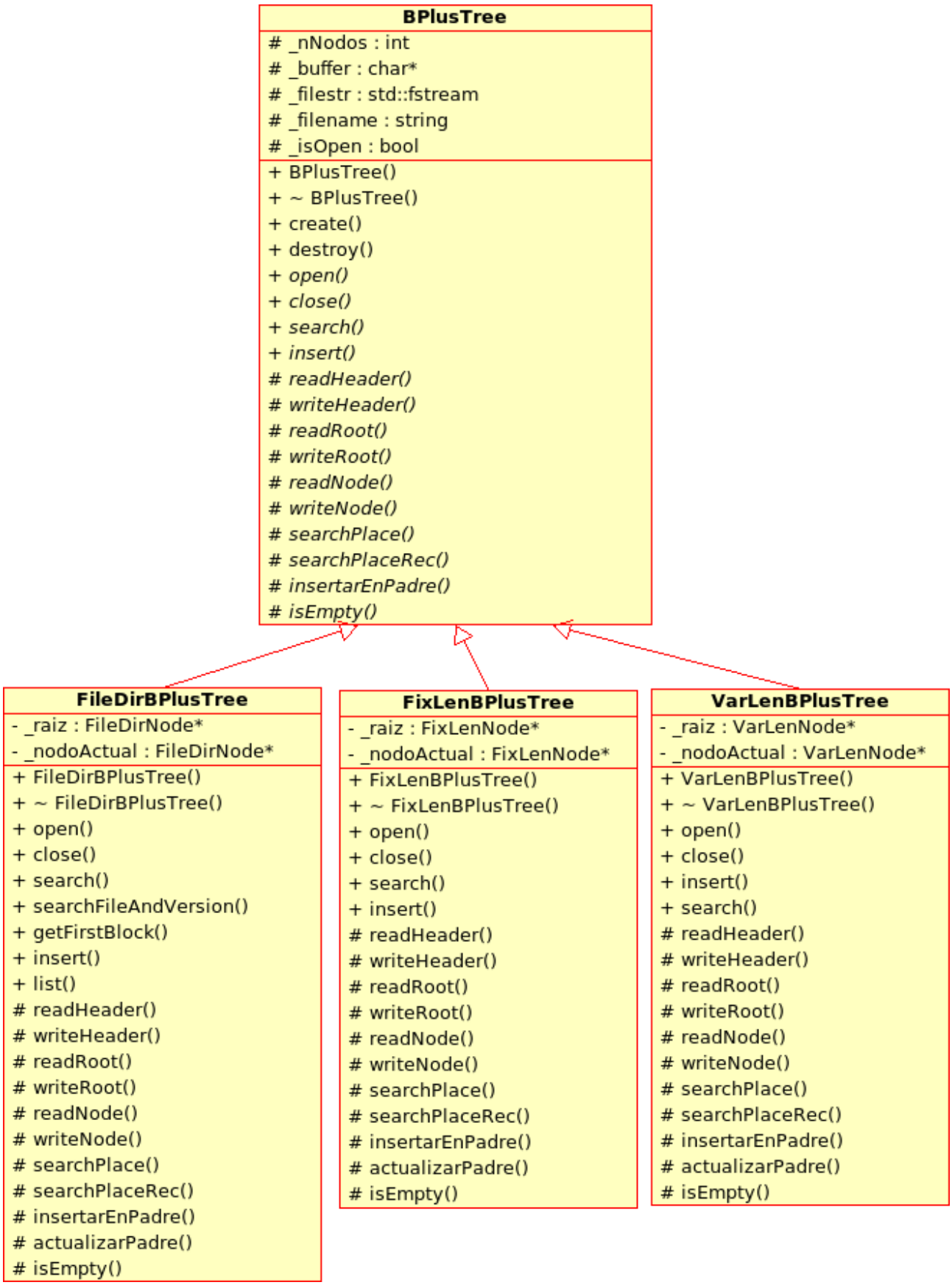
./svnuser usuario contraseña -o "nombre del repositorio" "nombre del directorio destino" "nombre del archivo o directorio" [version]
Si no se indica la versión, entonces se obtiene la ultima.

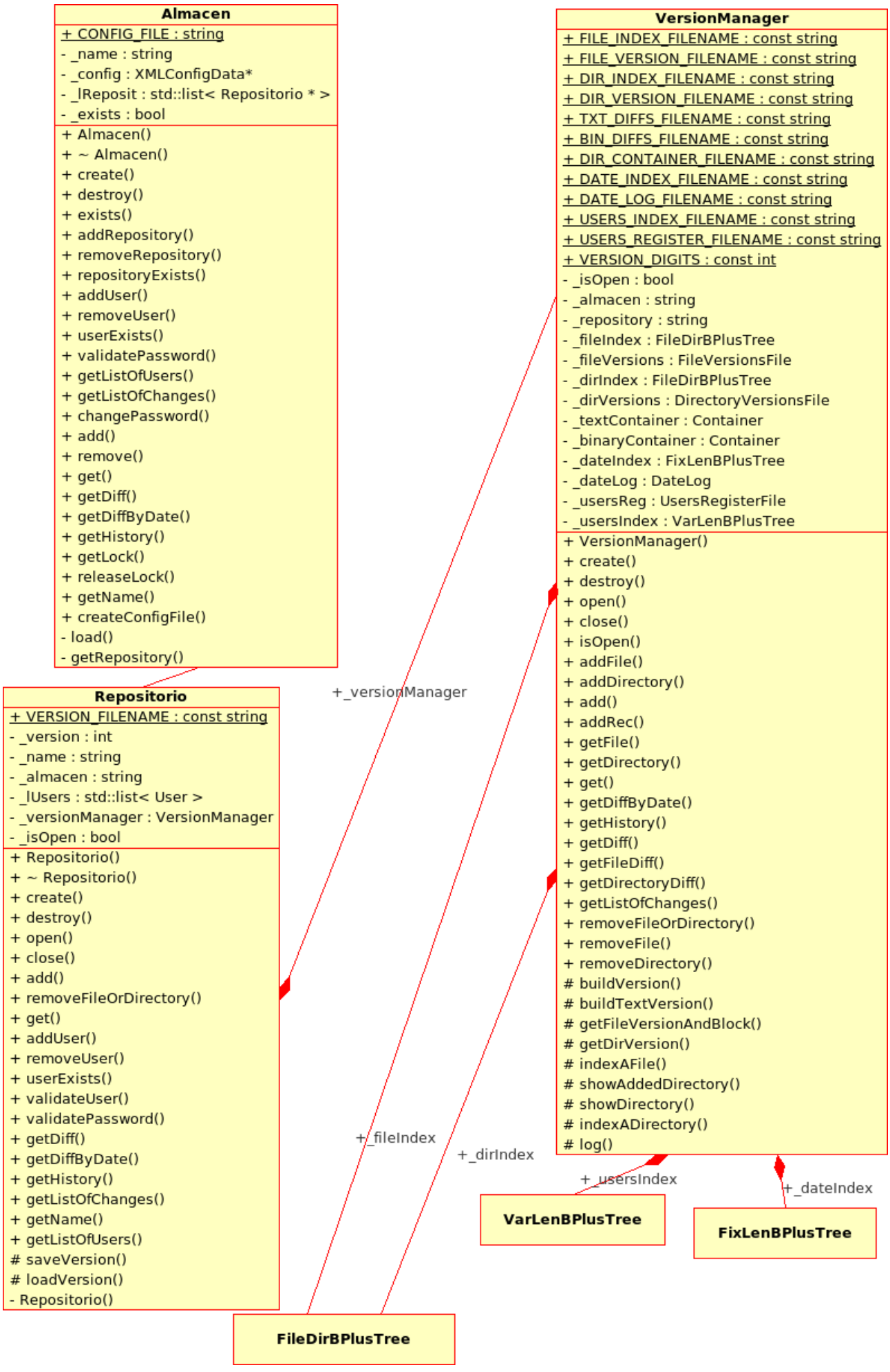
## Como cambiar la contraseña:

./svnuser usuario contraseña -p nueva "nombre del repositorio"

# Diagramas de clases


Node
# _nivel : int
# _id : int
# _padre : int
# _nclaves : int
+ Node()
+ ~ Node()
+ read()
+ write()
+ search()
+ isFull()
+ getId()
+ getPadre()
+ getCantClaves()
+ getNivel()
+ getRef()
+ getType()
+ setPadre()
# insertPair()
# readInfoAdm()
# writeInfoAdm()
# readDatos()
# writeDatos()
VarLenNode
+ VARLEN_NODE_SIZE : int
# _espacioLibre : int
# _offset : int
# _pares : char
+ VarLenNode()
+ ~ VarLenNode()
+ insert()
+ insert()
+ isFull()
+ getOffset()
+ getRef()
+ split()
+ promoteRoot()
# insertPair()
# insertPair()
# insertPair()
# completePairs()
# completePairs()
# readInfoAdm()
# writeInfoAdm()
# readDatos()
# writeDatos()
FixLenNode
+ FIXLEN_NODE_SIZE : int
+ MAXKEYS : int
# _pares : char
+ FixLenNode()
+ ~ FixLenNode()
+ search()
+ insert()
+ isFull()
+ getRef()
+ promoteRoot()
# insertPair()
# insertPair()
# completePairs()
# readInfoAdm()
# readDatos()
# writeInfoAdm()
# writeDatos()
FixLenLeafNode
# _hnoDerecho : int
# _hnoIzquierdo : int
+ FixLenLeafNode()
+ read()
+ write()
+ search()
+ promoteRoot()
+ convertirAIndice()
+ getHnoDerecho()
+ getHnoIzquierdo()
+ getType()
+ setHnoDerecho()
+ setHnoIzquierdo()
FixLenIndexNode
+ BYTESREDONDEO : const int
# _hijoIzquierdo : int
# _redondeo : char*
+ FixLenIndexNode()
+ read()
+ write()
+ search()
+ promoteRoot()
+ getHijoIzquierdo()
+ getType()
VarLenIndexNode
+ BYTESREDONDEO : const int
# _hijoIzquierdo : int
# _redondeo : char*
+ VarLenIndexNode()
+ ~ VarLenIndexNode()
+ read()
+ write()
+ search()
+ split()
+ promoteRoot()
+ getHijoIzquierdo()
+ getType()
VarLenLeafNode
- _hnoDerecho : int
- _hnoIzquierdo : int
+ VarLenLeafNode()
+ read()
+ write()
+ search()
+ split()
+ promoteRoot()
+ getHnoDerecho()
+ getHnoIzquierdo()
+ getType()
+ setHnoDerecho()
+ setHnoIzquierdo()
+ list()
FileDirNode
+ FileDirNode()
+ searchFileAndVersion()
+ getFirstOf()
FileDirIndexNode
+ BYTESREDONDEO : const int
# _hijoIzquierdo : int
# _redondeo : char*
+ FileDirIndexNode()
+ ~ FileDirIndexNode()
+ read()
+ write()
+ search()
+ searchFileAndVersion()
+ getFirstOf()
+ split()
+ promoteRoot()
+ getHijoIzquierdo()
+ getType()
FileDirLeafNode
- _hnoDerecho : int
- _hnoIzquierdo : int
+ FileDirLeafNode()
+ read()
+ write()
+ search()
+ searchFileAndVersion()
+ getFirstOf()
+ split()
+ promoteRoot()
+ getHnoDerecho()
+ getHnoIzquierdo()
+ getType()
+ setHnoDerecho()
+ setHnoIzquierdo()
+ list()


BPlusTree
# _nNodos : int
# _buffer : char*
# _filestr : std::fstream
# _filename : string
# _isOpen : bool
+ BPlusTree()
+ ~ BPlusTree()
+ create()
+ destroy()
+ open()
+ close()
+ search()
+ insert()
# readHeader()
# writeHeader()
# readRoot()
# writeRoot()
# readNode()
# writeNode()
# searchPlace()
# searchPlaceRec()
# insertarEnPadre()
# isEmpty()
FileDirBPlusTree
- _raiz : FileDirNode*
- _nodoActual : FileDirNode*
+ FileDirBPlusTree()
+ ~ FileDirBPlusTree()
+ open()
+ close()
+ search()
+ searchFileAndVersion()
+ getFirstBlock()
+ insert()
+ list()
# readHeader()
# writeHeader()
# readRoot()
# writeRoot()
# readNode()
# writeNode()
# searchPlace()
# searchPlaceRec()
# insertarEnPadre()
# actualizarPadre()
# isEmpty()
FixLenBPlusTree
- _raiz : FixLenNode*
- _nodoActual : FixLenNode*
+ FixLenBPlusTree()
+ ~ FixLenBPlusTree()
+ open()
+ close()
+ search()
+ insert()
# readHeader()
# writeHeader()
# readRoot()
# writeRoot()
# readNode()
# writeNode()
# searchPlace()
# searchPlaceRec()
# insertarEnPadre()
# actualizarPadre()
# isEmpty()
VarLenBPlusTree
- _raiz : VarLenNode*
- _nodoActual : VarLenNode*
+ VarLenBPlusTree()
+ ~ VarLenBPlusTree()
+ open()
+ close()
+ insert()
+ search()
# readHeader()
# writeHeader()
# readRoot()
# writeRoot()
# readNode()
# writeNode()
# searchPlace()
# searchPlaceRec()
# insertarEnPadre()
# actualizarPadre()
# isEmpty()


Almacen
+ CONFIG_FILE : string
- _name : string
- _config : XMLConfigData*
- _lReposit : std::list< Repositorio * >
- _exists : bool
+ Almacen()
+ ~ Almacen()
+ create()
+ destroy()
+ exists()
+ addRepository()
+ removeRepository()
+ repositoryExists()
+ addUser()
+ removeUser()
+ userExists()
+ validatePassword()
+ getListOfUsers()
+ getListOfChanges()
+ changePassword()
+ add()
+ remove()
+ get()
+ getDiff()
+ getDiffByDate()
+ getHistory()
+ getLock()
+ releaseLock()
+ getName()
+ createConfigFile()
- load()
- getRepository()
VersionManager
+ FILE_INDEX_FILENAME : const string
+ FILE_VERSION_FILENAME : const string
+ DIR_INDEX_FILENAME : const string
+ DIR_VERSION_FILENAME : const string
+ TXT_DIFFS_FILENAME : const string
+ BIN_DIFFS_FILENAME : const string
+ DIR_CONTAINER_FILENAME : const string
+ DATE_INDEX_FILENAME : const string
+ DATE_LOG_FILENAME : const string
+ USERS_INDEX_FILENAME : const string
+ USERS_REGISTER_FILENAME : const string
+ VERSION_DIGITS : const int
- _isOpen : bool
- _almacen : string
- _repository : string
- _fileIndex : FileDirBPlusTree
- _fileVersions : FileVersionsFile
- _dirIndex : FileDirBPlusTree
- _dirVersions : DirectoryVersionsFile
- _textContainer : Container
- _binaryContainer : Container
- _dateIndex : FixLenBPlusTree
- _dateLog : DateLog
- _usersReg : UsersRegisterFile
- _usersIndex : VarLenBPlusTree
+ VersionManager()
+ create()
+ destroy()
+ open()
+ close()
+ isOpen()
+ addFile()
+ addDirectory()
+ add()
+ addRec()
+ getFile()
+ getDirectory()
+ get()
+ getDiffByDate()
+ getHistory()
+ getDiff()
+ getFileDiff()
+ getDirectoryDiff()
+ getListOfChanges()
+ removeFileOrDirectory()
+ removeFile()
+ removeDirectory()
# buildVersion()
# buildTextVersion()
# getFileVersionAndBlock()
# getDirVersion()
# indexAFile()
# showAddedDirectory()
# showDirectory()
# indexADirectory()
# log()
Repositorio
+ VERSION_FILENAME : const string
- _version : int
- _name : string
- _almacen : string
- _lUsers : std::list< User >
- _versionManager : VersionManager
- _isOpen : bool
+ Repositorio()
+ ~ Repositorio()
+ create()
+ destroy()
+ open()
+ close()
+ add()
+ removeFileOrDirectory()
+ get()
+ addUser()
+ removeUser()
+ userExists()
+ validateUser()
+ validatePassword()
+ getDiff()
+ getDiffByDate()
+ getHistory()
+ getListOfChanges()
+ getName()
+ getListOfUsers()
# saveVersion()
# loadVersion()
- Repositorio()
+_versionManager
+_fileIndex
+_dirIndex
+_usersIndex
+_dateIndex
VarLenBPlusTree
FixLenBPlusTree
FileDirBPlusTree

# Código Fuente

**Node.h**

```
#ifndef NODE_H_INCLUDED
#define NODE_H_INCLUDED

#include <cstdlib>
#include <iostream>
#include <string>

class Node
{
public:
   enum t_status   { ALREADY_EXISTS = 0, OK, OVERFLOW, ERROR };
   enum t_nodeType { LEAF, INDEX };

   // constructors
   Node(int id = 0, int nivel = 0, int padre = -1);
   virtual ~Node() {};

   virtual void read (char* buffer) = 0;  // lee el nodo desde un buffer
   virtual void write(char* buffer) = 0;  // escribe el nodo en un buffer

   // busca la clave dentro del nodo, devuelve:
   // - una referencia al archivo de datos si la clave buscada esta en el nodo y este es un nodo hoja.
   // - -1 si la clave buscada no esta en el nodo y es un nodo hoja.
   // - la referencia al nodo siguiente donde buscar si el nodo es indice
   virtual int search(const char* key) = 0; // key = nombre del archivo a buscar

   virtual bool isFull() const = 0;

   // getters
   int getId()        const { return _id;      }
   int getPadre()     const { return _padre;   }
   int getCantClaves() const { return _nclaves; }
   int getNivel()     const { return _nivel;   }
   virtual int getRef(int idNode) = 0;

   virtual t_nodeType getType() const = 0;

   // setters
   void setPadre(int a_Padre) { _padre = a_Padre; }

protected:
   virtual bool insertPair(const char* key, int ref) = 0;

   virtual void readInfoAdm (char** nextByte) = 0;
   virtual void writeInfoAdm(char** nextByte) = 0;

   virtual void readDatos   (char** nextByte) = 0;
   virtual void writeDatos  (char** nextByte) = 0;

   // member variables
   int  _nivel;
   int  _id;
   int  _padre;        // referencia al padre del nodo
   int  _nclaves;
};

#endif
```

**Node.cpp**

```
#include "Node.h"

#include <iostream>

// constructor
Node::Node(int id, int nivel, int padre)
{
    _nivel        = nivel;
    _id           = id;
    _padre        = padre;
    _nclaves      = 0;
}
```

**FixLenNode.h**

```
#ifndef FIXLENNODE_H_INLCUDED
#define FIXLENNODE_H_INLCUDED

#include "Node.h"

#include <cstdlib>
#include <iostream>
#include <string>

#define FIXLEN_STREAM_SIZE 490
#define KEY_LENGTH 10

class FixLenNode : public Node
{
public:
  const static int FIXLEN_NODE_SIZE;
  const static int MAXKEYS;

  FixLenNode(int id = 0, int nivel = 0, int padre = -1);
  virtual ~FixLenNode() {};

  virtual int search(const char* key) = 0;

  t_status insert(const char* key, int ref);

  bool isFull() const { return _nclaves == MAXKEYS; }

  // getters
  int getRef(int idNode);

  // me genera 2 nuevos nodos al tener que promover la raiz
  virtual void promoteRoot(FixLenNode** nuevo_nodo,int id)  = 0;

protected:
  bool insertPair(const char* key, int ref);
  bool insertPair(const char* key, int ref, int* i, char* auxKey, int* auxRef);
  void completePairs(int* i, int* j, char* auxKey, int* auxRef);

  void readInfoAdm (char** nextByte);
  void readDatos   (char** nextByte);

  void writeInfoAdm(char** nextByte);
  void writeDatos  (char** nextByte);

  // member variables
  char  _pares[FIXLEN_STREAM_SIZE];// pares clave-referencia en el caso de los nodos hoja
                                   // pares clave-hijo claves mayores en el caso de los interiores
};

#endif
```

**FixLenNode.cpp**

```
#include "FixLenNode.h"

#include <iostream>

// CONSTS
const int FixLenNode::MAXKEYS           = 35;
const int FixLenNode::FIXLEN_NODE_SIZE  = 512;

// constructor
FixLenNode::FixLenNode(int id, int nivel, int padre) : Node(id, nivel, padre)
{}

bool FixLenNode::insertPair(const char* key, int ref, int* i, char* auxKey, int* auxRef)
{
   bool end = false;

   while (((*i) <_nclaves) && (!end) ) {
      // obtengo la clave y la referencia de la posicion i en la tira de bytes
      memcpy(auxKey, _pares+(( KEY_LENGTH+sizeof(int) )* (*i)), sizeof(char)*KEY_LENGTH);
      memcpy(auxRef, _pares+(( KEY_LENGTH+sizeof(int) )* (*i) + KEY_LENGTH * sizeof(char)), sizeof(int));

      int cmp = strcmp(key, auxKey);
      if (cmp < 0) {
         // inserto clave y referencia
         memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )* (*i)), key, sizeof(char)*KEY_LENGTH);
         memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )* (*i) + KEY_LENGTH * sizeof(char) ), &ref,
sizeof(int));
         end = true;
      }
      else if (cmp > 0)
         (*i)++;
      else {
         std::cout << "la clave que se intenta insertar ya existe" << std::endl;
         return false;
      }
   }
   if ((*i) >= _nclaves) {
      memcpy(auxKey, key, KEY_LENGTH * sizeof(char));
      memcpy(auxRef, &ref, sizeof(int));
   }
   return true;
}

void FixLenNode::completePairs(int* i,int* j,char auxKey[KEY_LENGTH],int* auxRef)
{
  (*i)++;
  for ((*j) = (*i); (*j) < _nclaves; ++(*j)) {
     char auxKey2[KEY_LENGTH];
     int auxRef2;

     memcpy(auxKey2, _pares+( (KEY_LENGTH * sizeof(char) + sizeof(int))* (*j)), sizeof(char) * KEY_LENGTH);
     memcpy(&auxRef2, _pares+( (KEY_LENGTH * sizeof(char) + sizeof(int))* (*j) + KEY_LENGTH * sizeof(char)),
sizeof(int));

     memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )* (*j)), auxKey, sizeof(char)*KEY_LENGTH);
     memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )* (*j) + KEY_LENGTH * sizeof(char) ), auxRef,
sizeof(int));
```

```
      strcpy(auxKey, auxKey2);
      *auxRef = auxRef2;
    }

  return;
}

bool FixLenNode::insertPair(const char* key, int ref)
{
  char auxKey[KEY_LENGTH];
  int auxRef, i = 0, j;

  if (insertPair(key, ref, &i, auxKey, &auxRef)) {
    if (i < _nclaves) {

      completePairs(&i, &j, auxKey, &auxRef);

      memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )*j), auxKey, sizeof(char)*KEY_LENGTH);

      memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )*j+ KEY_LENGTH * sizeof(char) ), &auxRef,
sizeof(int));
      _nclaves++;
    }

    else if ( i == _nclaves) {
      memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )*_nclaves), key, sizeof(char)*KEY_LENGTH);
      memcpy(_pares+( (KEY_LENGTH*sizeof(char) + sizeof(int) )*_nclaves + KEY_LENGTH * sizeof(char) ), &ref,
sizeof(int));
      _nclaves++;
    }
     return true;
   }
 return false;
}

void FixLenNode::readInfoAdm(char** nextByte)
{
   memcpy(&_nivel,*nextByte,sizeof(int));
   *nextByte += sizeof(int);
   memcpy(&_id,*nextByte,sizeof(int));
   *nextByte += sizeof(int);
   memcpy(&_padre,*nextByte,sizeof(int));
   *nextByte += sizeof(int);
}

void FixLenNode::readDatos(char** nextByte)
{
   memcpy(&_nclaves,*nextByte,sizeof(short int));
   *nextByte += sizeof(short int);
   memcpy(_pares,*nextByte,sizeof(char) * FIXLEN_STREAM_SIZE);
   *nextByte += sizeof(char) * FIXLEN_STREAM_SIZE;
}

void FixLenNode::writeInfoAdm(char** nextByte)
{
   memcpy(*nextByte,&_nivel,sizeof(int));
   *nextByte += sizeof(int);
   memcpy(*nextByte,&_id,sizeof(int));
   *nextByte += sizeof(int);
   memcpy(*nextByte,&_padre,sizeof(int));
```

```
    *nextByte += sizeof(int);
}

void FixLenNode::writeDatos(char** nextByte)
{
    memcpy(*nextByte,&_nclaves,sizeof(short int));
    *nextByte += sizeof(short int);
    memcpy(*nextByte,_pares,sizeof(char) * FIXLEN_STREAM_SIZE);
    *nextByte += sizeof(char) * FIXLEN_STREAM_SIZE;
}

FixLenNode::t_status FixLenNode::insert(const char* key, int ref)
{
   if (_nclaves < MAXKEYS) {
      if(insertPair(key,ref))
         return OK;

      return ERROR;
   }
   else
      return OVERFLOW;
}

int FixLenNode::getRef(int idNode)
{
    int ret;
    memcpy(&ret, _pares+( (KEY_LENGTH * sizeof(char) + sizeof(int))* idNode + KEY_LENGTH * sizeof(char)),
sizeof(int));
    return ret;
}
```

**FixLenIndexNode.h**

```
#ifndef FIXLENINDEXNODE_H_INCLUDED
#define FIXLENINDEXNODE_H_INCLUDED

#include "FixLenNode.h"

class FixLenIndexNode : public FixLenNode
{
public:
   static const int BYTESREDONDEO;

   // constructors
   FixLenIndexNode(int id = 0, int nivel = 0, int padre = -1, int HijoIzquierdo = 0, const char* key = 0, int ref = 0);

   void read(char* buffer);
   void write(char* buffer);

   int search(const char* key);

   void promoteRoot(FixLenNode** nuevo_nodo, int id);

   // getters
   int getHijoIzquierdo() const { return _hijoIzquierdo; }
   t_nodeType getType() const { return INDEX; }

protected:
   int   _hijoIzquierdo;            // la referencia al nodo con claves menores
   char* _redondeo; // bytes de redondeo para completar el tamanio del nodo en disco

};

#endif
```

**FixLenIndexNode.cpp**

```
#include "FixLenIndexNode.h"

const int FixLenIndexNode::BYTESREDONDEO = 4;

FixLenIndexNode::FixLenIndexNode(int id, int nivel, int padre, int HijoIzquierdo,const char* key, int ref) :
FixLenNode(id, nivel, padre)
{
   _redondeo = new char[BYTESREDONDEO];
   _hijoIzquierdo = HijoIzquierdo;

   if (BYTESREDONDEO > 0) {
      memcpy(_redondeo, "****",BYTESREDONDEO * sizeof(char));
      _redondeo[BYTESREDONDEO] = 0;
   }

   if (ref != 0) {
      insertPair(key, ref);
      _nclaves = 1;
   }
   else _nclaves = 0;
}

void FixLenIndexNode::read(char* buffer)
{
```

```
    char* nextByte = buffer;
    readInfoAdm(&nextByte);
    memcpy(&_hijoIzquierdo, nextByte, sizeof(int));
    nextByte += sizeof(int);
    readDatos(&nextByte);

    if (BYTESREDONDEO > 0)
        memcpy(&_redondeo, nextByte, sizeof(char) * BYTESREDONDEO);
}

void FixLenIndexNode::write(char* buffer)
{
    char* nextByte = buffer;
    writeInfoAdm(&nextByte);
    memcpy(nextByte, &_hijoIzquierdo, sizeof(int));
    nextByte += sizeof(int);
    writeDatos(&nextByte);

    if(BYTESREDONDEO > 0)
        memcpy(nextByte, &_redondeo, sizeof(char) * BYTESREDONDEO);
}

int FixLenIndexNode::search(const char* key)
{
   int ret = _hijoIzquierdo;
   char auxKey[KEY_LENGTH + 1];
   int auxRef;
   int i = 0;
            int offset = 0;
   while (i < _nclaves) {
      memcpy(auxKey, _pares+ offset,sizeof(char)*KEY_LENGTH);
                        auxKey[KEY_LENGTH] = 0;
                        offset += sizeof(char)*KEY_LENGTH;

      memcpy(&auxRef, _pares+ offset,sizeof(int));
                        offset += sizeof(int);

      if(strcmp(key, auxKey) < 0)
         return ret;
      else {
         ret = auxRef;
         i++;
      }
   }
  return ret;
}

void FixLenIndexNode::promoteRoot(FixLenNode** nuevo_nodo, int id)
{
   int i;
   int auxRef;
   char auxKey[KEY_LENGTH];
   *nuevo_nodo = new FixLenIndexNode(id,_nivel,_id,_hijoIzquierdo);
            int offset = 0;

   for(i = 0; i < MAXKEYS; ++i) {
      memcpy(auxKey, _pares + offset, sizeof(char)*KEY_LENGTH);
                        offset += sizeof(char)*KEY_LENGTH;

      memcpy(&auxRef, _pares + offset, sizeof(int));
```

```
                        offset += sizeof(int);
      (*nuevo_nodo)->insert(auxKey,auxRef);
    }

            _nclaves = 0;
            return;
}
```

**FixLenLeafNode.h**

```
#ifndef FIXLENLEAFNODE_H_INCLUDED
#define FIXLENLEAFNODE_H_INCLUDED

#include "FixLenNode.h"
#include "FixLenIndexNode.h"

class FixLenLeafNode : public FixLenNode
{
public:

        FixLenLeafNode(int id = 0, int padre = 0, int HnoIzquierdo = -1, int HnoDerecho = -1);

        void read (char* buffer);
        void write(char* buffer);

        int search(const char* key);

        void promoteRoot(FixLenNode** nuevo_nodo,int id);
        FixLenIndexNode* convertirAIndice(int HijoIzquierdo, int primerRef, const char* key);

   // getters
   int getHnoDerecho()   const { return _hnoDerecho; }
   int getHnoIzquierdo() const { return _hnoIzquierdo; }
   t_nodeType getType() const { return LEAF; }

   // setters
        void setHnoDerecho  (int HnoDerecho) { _hnoDerecho = HnoDerecho; }
        void setHnoIzquierdo(int HnoIzquierdo) { _hnoIzquierdo = HnoIzquierdo; }

protected:
        int _hnoDerecho;
        int _hnoIzquierdo;
};
#endif
```

**FixLenLeafNode.cpp**

```
#include "FixLenLeafNode.h"

FixLenLeafNode::FixLenLeafNode(int id, int padre, int HnoIzquierdo, int HnoDerecho) : FixLenNode(id, 0, padre)
{
   _hnoIzquierdo = HnoIzquierdo;
   _hnoDerecho   = HnoDerecho;
}

void FixLenLeafNode::read(char* buffer)
{
   char* nextByte = buffer;
   readInfoAdm(&nextByte);
   readDatos(&nextByte);
   memcpy(&_hnoIzquierdo, nextByte, sizeof(int));
   nextByte += sizeof(int);
   memcpy(&_hnoDerecho, nextByte, sizeof(int));
}

void FixLenLeafNode::write(char* buffer)
{
   char* nextByte = buffer;
```

```
    writeInfoAdm(&nextByte);
    writeDatos(&nextByte);
    memcpy(nextByte, &_hnoIzquierdo, sizeof(int));
    nextByte += sizeof(int);
    memcpy(nextByte, &_hnoDerecho, sizeof(int));
}

int FixLenLeafNode::search(const char* key)
{
   char auxKey[KEY_LENGTH + 1];
   int auxRef;
   int i = 0;

            int offset = 0;
   while (i < _nclaves) {
      memcpy(auxKey, _pares + offset, sizeof(char) * KEY_LENGTH);
                        auxKey[KEY_LENGTH] = 0;
                        offset += sizeof(char) * KEY_LENGTH;

      memcpy(&auxRef, _pares + offset, sizeof(int));
                        offset += sizeof(int);

      if (strcmp(key, auxKey) == 0)
           return auxRef;
      else
          i++;
   }

  return -1;
}

FixLenIndexNode* FixLenLeafNode::convertirAIndice(int HijoIzquierdo, int primerRef,const char* key)
{
    FixLenIndexNode* ret = new FixLenIndexNode(_id, _nivel + 1, _padre, HijoIzquierdo);

    ret->insert(key, primerRef);
    return ret;
}

void FixLenLeafNode::promoteRoot(FixLenNode** nuevo_nodo, int id)
{
   int i;
   int auxRef;
   char auxKey[KEY_LENGTH];

   *nuevo_nodo= new FixLenLeafNode(id, _id, -1,-1);

            int offset = 0;

   for (i = 0; i < MAXKEYS; ++i) {
      memcpy(auxKey, _pares + offset, sizeof(char) * KEY_LENGTH);
                        offset += sizeof(char) * KEY_LENGTH;

      memcpy(&auxRef, _pares + offset, sizeof(int));
                        offset += sizeof(int);

      (*nuevo_nodo)->insert(auxKey, auxRef);
    }
   return;
}
```

**VarLenNode.h**

```
#ifndef VARLENNODE_H_INCLUDED
#define VARLENNODE_H_INCLUDED

#include "Node.h"

#include <cstdlib>
#include <iostream>
#include <string>

#define VARLEN_STREAM_SIZE 2020

class VarLenNode : public Node
{
public:
  const static int VARLEN_NODE_SIZE;

  VarLenNode(int id = 0, int nivel = 0, int padre = -1);
  virtual ~VarLenNode() {};

  t_status insert(const char* key, int ref, int* clavesArreglo, char** arreglo, int* bytesArreglo);
  bool insert(const char* key, int ref) {
    return insertPair(key, ref);
  }

  bool isFull() const { return _espacioLibre == 0; }

  //getters
  int getOffset()   const { return _offset; }
  int getRef(int idNode);

  // generador de un nodo del mismo _nivel para cuando tengo que hacer un split
  virtual VarLenNode* split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char** claveAlPadre) = 0;

  // me genera 2 nuevos nodos al tener que promover la raiz
  virtual void promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2, int id1, int id2, int clavesArreglo, char**
arreglo, int bytesArreglo, char** claveARaiz)  = 0;

protected:
   bool insertPair(const char* key, int ref);
   bool insertPair(const char* key, int ref, int* offset, int* clavesArregloAux, char** arregloAux);
   bool insertPair(const char* key, int ref, int* clavesArreglo, char** arregloAux, int* bystesArreglo);

   void completePairs(int* offset, int tamanioArreglo, char* arregloAux);
   void completePairs(int* tamanioArreglo, char** arregloAux, int* bytesArreglo);

   void readInfoAdm (char** nextByte);
   void writeInfoAdm(char** nextByte);

   void readDatos   (char** nextByte);
   void writeDatos  (char** nextByte);

  //member variables
  int _espacioLibre;
  int _offset;
  char _pares[VARLEN_STREAM_SIZE];
};

#endif
```

**VarLenNode.cpp**

```
#include "VarLenNode.h"

#include <iostream>

const int VarLenNode::VARLEN_NODE_SIZE = 2048;  // arbitrario

// constructor
VarLenNode::VarLenNode(int id, int nivel, int padre) : Node(id, nivel, padre)
{
   _espacioLibre = VARLEN_STREAM_SIZE;
   _offset       = 0;
}

VarLenNode::t_status VarLenNode::insert(const char* key, int ref, int* clavesArreglo, char** arregloAux, int*
bytesArreglo)
{
   int tamanioClaveYRef = strlen(key) * sizeof(char) + sizeof(int) * 2;

            if (tamanioClaveYRef <= _espacioLibre)
      return (insertPair(key,ref) ? OK : ERROR);

            else if (insertPair(key, ref,clavesArreglo,arregloAux,bytesArreglo)) {
      completePairs(clavesArreglo, arregloAux,bytesArreglo);
      return OVERFLOW;  // se inserto normalmente pero con overflow
   }
   else
      return ALREADY_EXISTS; // no se pudo insertar porque la clave ya estaba
}

bool VarLenNode::insertPair(const char* key, int ref, int* offset, int* tamanioArregloAux, char** arregloAux)
{
   bool end = false;

            int tamanioClave;
            int offsetInsercion;
            int auxRef;
            char* auxKey;
            int tamanioClaveAInsertar = strlen(key);
            *tamanioArregloAux = _nclaves;

  while ((*offset < _offset) && !end) {
               // guardo el offset donde prodria llegar a insertar
                        offsetInsercion = *offset;
    // obtengo la clave y la referencia de la posicion i en la tira de bytes
                        // tomo el tamanio de la clave
                        memcpy(&tamanioClave,_pares + (*offset),sizeof(int));
                        *offset += sizeof(int);

                        // leo la clave
                        auxKey = new char[(tamanioClave + 1) * sizeof(char)];

                        memcpy(auxKey, _pares + (*offset), sizeof(char)*tamanioClave);
                        auxKey[tamanioClave] = 0;
                        *offset += tamanioClave * sizeof(char);

                        // leo la referencia
    memcpy(&auxRef, _pares + (*offset), sizeof(int));
                        *offset += sizeof(int);
```

```
                // genero el nombre del archivo que esta en el nodo
                char* auxName = new char[(tamanioClave - 4) * sizeof(char)];
                memcpy(auxName,auxKey,(tamanioClave - 5));
                auxName[tamanioClave - 5] = 0;

                //genero la version string del archivo que esta en el nodo
                char* strNumber = new char[6];
                memcpy(strNumber, auxKey + (tamanioClave - 5),5*sizeof(char));
                strNumber[5] = 0;

                int auxNumber = atoi(strNumber);

                delete strNumber;

                int keySize = strlen(key);

                // genero el nombre del archivo que esta en el nodo
                char* auxName2 = new char[(keySize - 4) * sizeof(char)];
                memcpy(auxName2,key,(keySize - 5));
                auxName2[keySize - 5] = 0;

                //genero la version string del archivo que esta en el nodo
                char* strNumber2 = new char[6];
                memcpy(strNumber2, key + (keySize - 5),5*sizeof(char));
                strNumber2[5] = 0;

                int auxNumber2 = atoi(strNumber2);

                delete(strNumber2);

int cmp_filename = strcmp(auxName2,auxName);

                delete auxName;
                delete auxName2;

if (cmp_filename < 0) {
  // genero un arreglo auxiliar para poder contemplar los corrimientos en el arreglo
  // de claves
  *arregloAux = new char[(VARLEN_STREAM_SIZE - offsetInsercion) * sizeof(char)];
                        memcpy(*arregloAux,_pares + offsetInsercion,VARLEN_STREAM_SIZE - offsetInsercion);
                        // inserto tamanio de la clave, clave y referencia
                        // tamanio clave
                        memcpy(_pares + offsetInsercion,  &tamanioClaveAInsertar, sizeof(int));
                        offsetInsercion += sizeof(int);
                        // clave
  memcpy(_pares + offsetInsercion, key, sizeof(char) * tamanioClaveAInsertar);
                        offsetInsercion += tamanioClaveAInsertar * sizeof(char);
                        // referencia
  memcpy(_pares + offsetInsercion, &ref, sizeof(int));
                        offsetInsercion += sizeof(int);
                        *offset = offsetInsercion;
  end = true;
}
        else if(cmp_filename == 0)
                {
                  if(auxNumber2 < auxNumber)
                  {
                    // genero un arreglo auxiliar para poder contemplar los corrimientos en el arreglo
                    // de claves
```

```
                        *arregloAux = new char[(VARLEN_STREAM_SIZE - offsetInsercion) * sizeof(char)];
                                memcpy(*arregloAux,_pares + offsetInsercion,VARLEN_STREAM_SIZE -
offsetInsercion);
                                // inserto tamanio de la clave, clave y referencia
                                        // tamanio clave
                                        memcpy(_pares + offsetInsercion, &tamanioClaveAInsertar, sizeof(int));
                                        offsetInsercion += sizeof(int);
                                        // clave
                        memcpy(_pares + offsetInsercion, key, sizeof(char) * tamanioClaveAInsertar);
                                        offsetInsercion += tamanioClaveAInsertar * sizeof(char);
                                        // referencia
                        memcpy(_pares + offsetInsercion, &ref, sizeof(int));
                                offsetInsercion += sizeof(int);
                                *offset = offsetInsercion;
                        end = true;
                                }
                                else if(auxNumber == auxNumber2){
                                  std::cout << "la clave que se intenta insertar ya existe" << std::endl;
        return false;
                                }
                        }

                        if(!end)
                                (*tamanioArregloAux)--;

                        delete auxKey;
  }
  if (!end) { // si llego aca es porque tengo que insertar al final
    // inserto tamanio de la clave, clave y referencia
    // tamanio clave
    memcpy(_pares + (*offset), &tamanioClaveAInsertar, sizeof(int));
    *offset += sizeof(int);
    // clave
    memcpy(_pares + *offset, key, sizeof(char)*tamanioClaveAInsertar);
    *offset += tamanioClaveAInsertar * sizeof(char);
    // referencia
    memcpy(_pares + *offset, &ref, sizeof(int));
    *offset += sizeof(int);

    _offset = *offset;
  }
  return true;
}

bool VarLenNode::insertPair(const char* key, int ref, int *clavesArreglo, char** arregloAux, int* bytesArreglo)
{
        int offset = 0;
        int cantidadClaves = 0;
  bool end = false;

        int tamanioClave;
        int offsetInsercion;
        char* auxKey;
        int auxRef;
        int tamanioClaveAInsertar = strlen(key);
        *clavesArreglo = _nclaves + 1;

   while ((offset < _offset) && !end) {
                        // guardo el offser donde prodria llegar a insertar
                        offsetInsercion = offset;
```

```
    // obtengo la clave y la referencia de la posicion i en la tira de bytes
                    // tomo el tamanio de la clave
                    memcpy(&tamanioClave,_pares + offset,sizeof(int));
                    offset += sizeof(int);
                    // leo la clave
                    auxKey = new char[(tamanioClave + 1) * sizeof(char)];
                    memcpy(auxKey, _pares + offset, sizeof(char)*tamanioClave);
                    auxKey[tamanioClave] = 0;
                    offset += tamanioClave * sizeof(char);
                    // leo la referencia
    memcpy(&auxRef, _pares + offset,sizeof(int));
                    offset += sizeof(int);

    int cmp = strcmp(key,auxKey);
    if (cmp < 0) {
      // genero un arreglo auxiliar para poder contemplar los corrimientos en el arreglo
      // de claves
                            *bytesArreglo = VARLEN_STREAM_SIZE - offsetInsercion + tamanioClaveAInsertar *
sizeof(char) + 2 * sizeof(int);
      *arregloAux = new char[(*bytesArreglo) * sizeof(char)];
                            int offsetArreglo = 0;

                            // copio la clave que quiero insertar en el arreglo junto con los datos administrativos
                            // inserto tamaño de la clave, clave y referencia
                            // tamaño clave
                            memcpy(*arregloAux + offsetArreglo, &tamanioClaveAInsertar,         sizeof(int));
                            offsetArreglo += sizeof(int);

                            // clave
      memcpy(*arregloAux + offsetArreglo, key, sizeof(char)*tamanioClaveAInsertar);
                            offsetArreglo += tamanioClaveAInsertar * sizeof(char);
                            // referencia
      memcpy(*arregloAux + offsetArreglo, &ref, sizeof(int));
                            offsetArreglo += sizeof(int);
                            // copio el resto del arreglo que le sigue a la clave nueva
                            memcpy(*arregloAux + offsetArreglo, _pares + offsetInsercion, VARLEN_STREAM_SIZE -
offsetInsercion);
      end = true;
    }

    else if (cmp == 0) {
      std::cout << "la clave que se intenta insertar ya existe" << std::endl;
      return false;
    }
                    if (!end) {
                            (*clavesArreglo)--;
                            cantidadClaves++;
                    }
            delete auxKey;
  }

          if (!end) {
      *bytesArreglo = tamanioClaveAInsertar * sizeof(char) + 2 * sizeof(int);
      *arregloAux = new char[(*bytesArreglo) * sizeof(char)];
      int offsetArreglo = 0;

      // copio la clave que quiero insertar en el arreglo junto con los datos administrativos
      // inserto tamaño de la clave, clave y referencia
      // tamaño clave
      memcpy(*arregloAux + offsetArreglo,
```

```
                &tamanioClaveAInsertar,
                sizeof(int));
        offsetArreglo += sizeof(int);
        // clave
        memcpy(*arregloAux + offsetArreglo, key, sizeof(char)*tamanioClaveAInsertar);
        offsetArreglo += tamanioClaveAInsertar * sizeof(char);
        // referencia
        memcpy(*arregloAux + offsetArreglo, &ref, sizeof(int));
        offsetArreglo += sizeof(int);
            }

            if (end)
                    _offset = offsetInsercion;

            _nclaves = cantidadClaves;
            _espacioLibre = VARLEN_STREAM_SIZE - _offset;

            return true;
}

bool VarLenNode::insertPair(const char* key, int ref)
{
            char* arregloAux = 0;
  int offset = 0,
  tamanioArregloAux = 0;

  if (insertPair(key, ref, &offset,&tamanioArregloAux,&arregloAux)) {
    if (tamanioArregloAux != 0) {
      completePairs(&offset,tamanioArregloAux,arregloAux);
                        delete arregloAux;
                    }
                    _nclaves++;
                    _espacioLibre -= (strlen(key) * sizeof(char) + 2 * sizeof(int));
                    _offset = VARLEN_STREAM_SIZE - _espacioLibre;
    return true;
  }
  return false;
}

void VarLenNode::completePairs(int* offset, int tamanioArreglo, char* arregloAux)
{
            char* nextByte = arregloAux;

            char* auxKey;
            int auxRef;
            int tamanioClave;
   for (int i = 0; i < tamanioArreglo; ++i) {

                    // leo del arregloAuxiliar todos los valores que voy a copiar
                    memcpy(&tamanioClave,nextByte,sizeof(int));
                    nextByte += sizeof(int);

                    auxKey = new char[tamanioClave * sizeof(char)];
                    memcpy(auxKey,nextByte,tamanioClave * sizeof(char));
                    nextByte += tamanioClave * sizeof(char);
                    memcpy(&auxRef,nextByte,sizeof(int));
                    nextByte += sizeof(int);

                    //copio dentro del arreglo del nodo la informacion
                    memcpy(_pares + (*offset),&tamanioClave,sizeof(int));
```

```
                (*offset) += sizeof(int);
                memcpy(_pares + (*offset),auxKey,tamanioClave * sizeof(char));
                (*offset) += tamanioClave * sizeof(char);
                memcpy(_pares + (*offset),&auxRef,sizeof(int));
                (*offset) += sizeof(int);

                delete(auxKey);
    }
}

void VarLenNode::completePairs(int* tamanioArreglo, char **arregloAux, int* bytesArreglo)
{
        char* arregloAux2;
        char* auxKey;
        int tamanioClaveAux;
        int auxRef;

        int offsetArreglo = 0;

        while (VARLEN_STREAM_SIZE / 2 <= _espacioLibre) {

                // levanto tamanio de la clave, clave y referencia
                memcpy(&tamanioClaveAux,*arregloAux + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                auxKey = new char[tamanioClaveAux * sizeof(char)];
                memcpy(auxKey,*arregloAux + offsetArreglo,tamanioClaveAux * sizeof(char));
                offsetArreglo += tamanioClaveAux * sizeof(char);

                memcpy(&auxRef,*arregloAux + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                (*tamanioArreglo)--;

                // ahora copio en el arreglo del nodo estos datos
                memcpy(_pares + _offset,&tamanioClaveAux,sizeof(int));
                _offset += sizeof(int);

                memcpy(_pares + _offset,auxKey,tamanioClaveAux * sizeof(char));
                _offset += tamanioClaveAux * sizeof(char);

                memcpy(_pares + _offset,&auxRef,sizeof(int));
                _offset += sizeof(int);

                _nclaves++;
                _espacioLibre = VARLEN_STREAM_SIZE - _offset;

                delete auxKey;
        }

        int bytesACopiar = *bytesArreglo - offsetArreglo;
        arregloAux2 = new char[bytesACopiar * sizeof(char)];
        memcpy(arregloAux2,*arregloAux + offsetArreglo,bytesACopiar);
        delete(*arregloAux);
        *arregloAux = arregloAux2;
        *bytesArreglo = bytesACopiar;
}

void VarLenNode::readInfoAdm(char** nextByte)
{
```

```
    memcpy(&_nivel,*nextByte,sizeof(int));
    *nextByte += sizeof(int);

    memcpy(&_id,*nextByte,sizeof(int));
    *nextByte += sizeof(int);

    memcpy(&_padre,*nextByte,sizeof(int));
    *nextByte += sizeof(int);

            memcpy(&_espacioLibre,*nextByte,sizeof(int));
            *nextByte += sizeof(int);

            _offset = VARLEN_STREAM_SIZE - _espacioLibre;
}

void VarLenNode::readDatos(char** nextByte)
{
    memcpy(&_nclaves,*nextByte,sizeof(_nclaves));
    *nextByte += sizeof(_nclaves);
    memcpy(_pares,*nextByte,sizeof(char) * VARLEN_STREAM_SIZE);
    *nextByte += sizeof(char) * VARLEN_STREAM_SIZE;
}

void VarLenNode::writeInfoAdm(char** nextByte)
{
    memcpy(*nextByte,&_nivel,sizeof(int));
    *nextByte += sizeof(int);
    memcpy(*nextByte,&_id,sizeof(int));
    *nextByte += sizeof(int);
    memcpy(*nextByte,&_padre,sizeof(int));
    *nextByte += sizeof(int);
            memcpy(*nextByte,&_espacioLibre,sizeof(int));
            *nextByte += sizeof(int);
}

void VarLenNode::writeDatos(char** nextByte)
{
    memcpy(*nextByte,&_nclaves,sizeof(_nclaves));
    *nextByte += sizeof(_nclaves);
    memcpy(*nextByte,_pares,sizeof(char) * VARLEN_STREAM_SIZE);
    *nextByte += sizeof(char) * VARLEN_STREAM_SIZE;
}

int VarLenNode::getRef(int idNode)
{
    int ret = 0;
            int tamanioClave = 0;
            int offsetArreglo = 0;

            for (int i = 0; i < idNode; ++i) {
                    memcpy(&tamanioClave, _pares + offsetArreglo, sizeof(int));
                    offsetArreglo += sizeof(int) + tamanioClave * sizeof(char);
                    memcpy(&ret, _pares + offsetArreglo, sizeof(int));
                    offsetArreglo += sizeof(int);
            }
    return ret;
}
```

**VarLenIndexNode.h**

```
#ifndef VARLENINDEXNODE_H_INCLUDED
#define VARLENINDEXNODE_H_INCLUDED

#include "VarLenNode.h"

class VarLenIndexNode : public VarLenNode
{
public:
  static const int BYTESREDONDEO;

  // constructors
  VarLenIndexNode(int id = 0, int nivel = 0, int padre = -1, int HijoIzquierdo = 0, char* key = 0, int ref = 0);
        virtual ~VarLenIndexNode();

  void read(char* buffer);
  void write(char* buffer);

  int search(const char* key);

  VarLenNode* split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char** claveAlPadre);

  void promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2, int id1, int id2, int clavesArreglo, char** arreglo, int
bytesArreglo,
  char** claveARaiz);

  // getters
  int getHijoIzquierdo() const { return _hijoIzquierdo; }
  t_nodeType getType() const { return INDEX; }

protected:
   int   _hijoIzquierdo;          // la referencia al nodo con claves menores
   char* _redondeo; // bytes de redondeo para completar el tamanio del nodo en disco
};

#endif
```

**VarLenIndexNode.cpp**

```
#include "VarLenIndexNode.h"

#include <string>

const int VarLenIndexNode::BYTESREDONDEO = 2;  // (_hnoIzquierdo + _hnoDerecho) (nodobmashoja) -
_hijoIzquierdo (nodobmasindice)

VarLenIndexNode::VarLenIndexNode(int id, int nivel, int padre, int HijoIzquierdo, char* key, int ref) : VarLenNode(id,
nivel, padre)
{
  _hijoIzquierdo = HijoIzquierdo;

  if (BYTESREDONDEO > 0) {
    _redondeo = new char[BYTESREDONDEO + 1];
    strcpy(_redondeo, std::string(BYTESREDONDEO, '*').c_str());
    _redondeo[BYTESREDONDEO] = 0;
  }
  if (ref != 0) {
    insertPair(key, ref);
    _nclaves = 1;
```

```
  }

  else _nclaves = 0;
}

VarLenIndexNode::~VarLenIndexNode()
{
  delete _redondeo;
}

void VarLenIndexNode::read(char* buffer)
{
  char* nextByte = buffer;
  readInfoAdm(&nextByte);
  memcpy(&_hijoIzquierdo, nextByte, sizeof(int));
  nextByte += sizeof(int);
  readDatos(&nextByte);

  if (BYTESREDONDEO > 0)
    memcpy(&_redondeo, nextByte, sizeof(char) * BYTESREDONDEO);
}

void VarLenIndexNode::write(char* buffer)
{
  char* nextByte = buffer;
  writeInfoAdm(&nextByte);
  memcpy(nextByte, &_hijoIzquierdo, sizeof(int));
  nextByte += sizeof(int);
  writeDatos(&nextByte);

  if(BYTESREDONDEO > 0)
    memcpy(nextByte, &_redondeo, sizeof(char) * BYTESREDONDEO);
}

int VarLenIndexNode::search(const char* key)
{
  int ret = _hijoIzquierdo;

  char* auxKey;
        char* auxName;
  int auxRef;
        int tamanioClave;
        int offset = 0;

        if(_nclaves){
    int i = 0;
                while (i < _nclaves) {

                        memcpy(&tamanioClave,_pares + offset,sizeof(int));
                        offset += sizeof(int);

                        auxKey = new char[(tamanioClave + 1) * sizeof(char)];
                        memcpy(auxKey, _pares + offset,sizeof(char)*tamanioClave);
                        auxKey[tamanioClave] = 0;
                        offset += tamanioClave * sizeof(char);

                        auxName = new char[(tamanioClave - 4) * sizeof(char)];
                        memcpy(auxName,auxKey,(tamanioClave - 5));
                        auxName[tamanioClave - 5] = 0;
```

```
                        memcpy(&auxRef, _pares + offset, sizeof(int));
                        offset += sizeof(int);

                        if(strcmp(key, auxName) < 0){
                                delete auxKey;
                                delete auxName;
                                return ret;
                        }
                        else{
                                ret = auxRef;
                                i++;
                                delete auxKey;
                                delete auxName;
                        }
                }
                return ret;
        }
        return ret;
}

VarLenNode* VarLenIndexNode::split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char**
claveAlPadre)
{
        VarLenIndexNode* ret;
  int auxRef;
  char* auxKey;
        int longClave;
        int offsetArreglo = 0;

        int clavesCopiadas = 0;

        // si tengo 1 sola clave en el arreglo trato de poder sacar alguna del nodo actual
        // para poder insertarla en el nuevo nodo
        if(( clavesArreglo == 1)&&(_espacioLibre < VARLEN_STREAM_SIZE /2)){
    int clavesLeidas = 0;

                while(offsetArreglo < VARLEN_STREAM_SIZE /2){
                        // tamanio de la clave
                        memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
                        offsetArreglo += sizeof(int);
                        // avanzo la cantidad de caracteres de la clave
                        offsetArreglo += sizeof(char) * longClave;
                        // avanzo la cantidad de bytes que ocupa la referencia
                        offsetArreglo += sizeof(int);
                        clavesLeidas++;
                }

                // nuevo offset en el arreglo del nodo actual
                int nuevoOffset = offsetArreglo;
                // copio a partir de ahora todas las claves que restan al nuevo nodo
                for(int i = clavesLeidas; i < _nclaves;i++){
                        memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
                        offsetArreglo += sizeof(int);

                        auxKey = new char[(longClave +1) * sizeof(char)];
                        memcpy(auxKey,_pares + offsetArreglo,sizeof(char)*longClave);
                        auxKey[longClave] = 0;
                        offsetArreglo += sizeof(char)*longClave;

                        memcpy(&auxRef,_pares + offsetArreglo, sizeof(int));
```

```
                        offsetArreglo += sizeof(int);

                        if(!clavesCopiadas)
                        {
                                *claveAlPadre = new char[(longClave + 1) * sizeof(char)];
                                strcpy(*claveAlPadre,auxKey);
                                ret = new VarLenIndexNode(Numero,_nivel,_padre,auxRef);
                        }
                        else
                                ret->insert(auxKey,auxRef);

                        delete(auxKey);
                        clavesCopiadas++;
                }

                _offset = nuevoOffset;
                _nclaves = clavesLeidas;
                _espacioLibre = VARLEN_STREAM_SIZE - _offset;
        }

        offsetArreglo = 0;
        // sino la 1º clave del nuevo nodo va a ser la que viene desde el arreglo
  int copiadasDelArreglo = 0;

        if(!clavesCopiadas){
    // tomo la clave menor para subir al _padre
                memcpy(&longClave, arreglo + offsetArreglo, sizeof(int));
                offsetArreglo += sizeof(int);
                *claveAlPadre = new char[(longClave + 1)*sizeof(char)];
                memcpy(*claveAlPadre,arreglo + offsetArreglo,longClave * sizeof(char));
                (*claveAlPadre)[longClave] = 0;
                offsetArreglo += longClave * sizeof(char);

                // tomo la referencia para poder setear el hijo izquierdo
                memcpy(&auxRef,arreglo + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                ret = new VarLenIndexNode(Numero,_nivel,_padre,auxRef);
                copiadasDelArreglo++;
        }

    // copio las claves restantes al nuevo nodo
    for (int j = copiadasDelArreglo; j < clavesArreglo; j++){
      memcpy(&longClave,arreglo + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                auxKey = new char[(longClave + 1)*sizeof(char)];
                memcpy(auxKey, arreglo + offsetArreglo, sizeof(char)*longClave);
                auxKey[longClave] = 0;
                offsetArreglo += longClave * sizeof(char);

      memcpy(&auxRef, arreglo + offsetArreglo, sizeof(int));
                offsetArreglo += sizeof(int);

      ret->insert(auxKey,auxRef);
                delete auxKey;
    }

    return ret;
}
```

```
void VarLenIndexNode::promoteRoot(VarLenNode** nodo1,VarLenNode** nodo2,int id1,int id2, int
clavesArreglo,char** arreglo,int bytesArreglo,
                                                                    char** claveARaiz)
{
  int i;
  int auxRef;
  char* auxKey;
          int tamanioClave;
          int offsetArreglo = 0;

  *nodo1 = new VarLenIndexNode(id1,_nivel,_id,_hijoIzquierdo);

          // copio todas las claves de la raiz al nodo1
  for(i = 0; i < _nclaves; ++i) {

    memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
                    offsetArreglo += sizeof(int);

                    auxKey = new char[(tamanioClave +1) * sizeof(char)];
                    memcpy(auxKey, _pares + offsetArreglo, sizeof(char)*tamanioClave);
                    auxKey[tamanioClave] = 0;
                    offsetArreglo += tamanioClave * sizeof(char);

    memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
                    offsetArreglo += sizeof(int);

    (*nodo1)->insert(auxKey,auxRef);

                    delete(auxKey);
  }

          // completo hasta tener la mitad del nodo1 "ocupada"
          offsetArreglo = 0;
          int clavesCopiadas = 0;

          while((*nodo1)->getOffset() < VARLEN_STREAM_SIZE / 2)
          {
            memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
                    offsetArreglo += sizeof(int);

                    auxKey = new char[(tamanioClave +1)*sizeof(char)];
                    memcpy(auxKey,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
                    auxKey[tamanioClave] = 0;
                    offsetArreglo += tamanioClave * sizeof(char);

                    memcpy(&auxRef,(*arreglo) + offsetArreglo,sizeof(int));
                    offsetArreglo += sizeof(int);

                    (*nodo1)->insert(auxKey,auxRef);

                    delete(auxKey);

                    clavesCopiadas++;
  }

  // tomo la clave que va a ir a la raiz nueva y
          // la referencia que va a ir al hijo izquierdo del nodo 2
          memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
          offsetArreglo += sizeof(int);
```

```
        *claveARaiz = new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(*claveARaiz,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
        (*claveARaiz)[tamanioClave] = 0;
        offsetArreglo += sizeof(char)*tamanioClave;

        memcpy(&auxRef,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        clavesCopiadas++;

        // creo el nodo 2 con el hijo izquierdo apuntando a auxRef
  *nodo2 = new VarLenIndexNode(id2,_nivel,_id,auxRef);

        //copio las claves y referencias restantes del arreglo
  for (i = clavesCopiadas; i < clavesArreglo; ++i) {
    memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                auxKey = new char[(tamanioClave + 1)*sizeof(char)];
                memcpy(auxKey, (*arreglo) + offsetArreglo, sizeof(char)*tamanioClave);
                auxKey[tamanioClave] = 0;
                offsetArreglo += sizeof(char)*tamanioClave;

    memcpy(&auxRef, (*arreglo) + offsetArreglo, sizeof(int));
                offsetArreglo += sizeof(int);

    (*nodo2)->insert(auxKey,auxRef);

                delete (auxKey);
                clavesCopiadas++;
  }

}
```

**VarLenLeafNode.h**

```
#ifndef VARLENLEAFNODE_H_INCLUDED
#define VARLENLEAFNODE_H_INCLUDED

#include "VarLenNode.h"
#include "VarLenIndexNode.h"

class VarLenLeafNode : public VarLenNode
{
public:
   VarLenLeafNode(int id = 0, int padre = 0, int HnoIzquierdo = -1, int HnoDerecho = -1);

   void read (char* buffer);
   void write(char* buffer);

   int search(const char* key);

   VarLenNode* split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char** claveAlPadre);

   void promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2, int id1, int id2, int clavesArreglo, char** arreglo, int
bytesArreglo,
         char** claveARaiz);

   // getters
   int getHnoDerecho()   const { return _hnoDerecho; }
   int getHnoIzquierdo() const { return _hnoIzquierdo; }
   t_nodeType getType() const { return LEAF; }

   // setters
   void setHnoDerecho  (int HnoDerecho) { _hnoDerecho = HnoDerecho; }
   void setHnoIzquierdo(int HnoIzquierdo) { _hnoIzquierdo = HnoIzquierdo; }

   // lista las claves y referencias del nodo
   void list();

private:
         int _hnoDerecho;
         int _hnoIzquierdo;
};

#endif
```

**VarLenLeafNode.cpp**

```
#include "VarLenLeafNode.h"

using std::cout;
using std::endl;

VarLenLeafNode::VarLenLeafNode(int id, int padre, int HnoIzquierdo, int HnoDerecho) : VarLenNode(id, 0, padre)
{
   _hnoIzquierdo = HnoIzquierdo;
   _hnoDerecho   = HnoDerecho;
}

void VarLenLeafNode::read(char* buffer)
{
   char* nextByte = buffer;
   readInfoAdm(&nextByte);
```

```
    readDatos(&nextByte);
    memcpy(&_hnoIzquierdo, nextByte, sizeof(int));
    nextByte += sizeof(int);
    memcpy(&_hnoDerecho, nextByte, sizeof(int));
}

void VarLenLeafNode::write(char* buffer)
{
    char* nextByte = buffer;
    writeInfoAdm(&nextByte);
    writeDatos(&nextByte);
    memcpy(nextByte, &_hnoIzquierdo, sizeof(int));
    nextByte += sizeof(int);
    memcpy(nextByte, &_hnoDerecho, sizeof(int));
}

int VarLenLeafNode::search(const char* key)
{
    char* auxKey;
    char* auxName;
    int longClave;
    int auxRef;
    int i = 0;
    int offsetLectura = 0;
    int ret = -1;

    while (i < _nclaves) {
        memcpy(&longClave,_pares + offsetLectura,sizeof(int));
        offsetLectura += sizeof(int);

        auxKey = new char[(longClave + 1) * sizeof(char)];

        memcpy(auxKey, _pares + offsetLectura, sizeof(char) * longClave);
        offsetLectura += sizeof(char) * longClave;
        auxKey[longClave] = 0; // coloco la marca de fin

        // genero el nombre del archivo
        auxName = new char[(longClave - 4) * sizeof(char)];
        memcpy(auxName,auxKey,(longClave - 5));
        auxName[longClave - 5] = 0;

        memcpy(&auxRef, _pares + offsetLectura, sizeof(int));
        offsetLectura += sizeof(int);

        if (strcmp(key, auxName) == 0){
            i++;
            ret = auxRef;
            delete auxKey;
            delete auxName;
        }
        else if(strcmp(key,auxName) > 0)
        {
            i++;
            delete auxKey;
            delete auxName;
        }
        else{
            delete auxName;
            delete(auxKey);
            return ret;
```

```
        }
    }
    return ret;
}

VarLenNode* VarLenLeafNode::split(int Numero, char* arreglo,int bytesArreglo,int clavesArreglo, char**
claveAlPadre)
{
    int longClave = 0;
    int offsetArreglo = 0;
    VarLenLeafNode* ret = new VarLenLeafNode(Numero, _padre, _id, _hnoDerecho);

    int auxRef;
    char* auxKey;
    int clavesCopiadas = 0;

    // si tengo 1 sola clave en el arreglo trato de poder sacar alguna del nodo actual
    // para poder insertarla en el nuevo nodo
    if(( clavesArreglo == 1)&&(_espacioLibre < VARLEN_STREAM_SIZE /2)){

        int clavesLeidas = 0;

        while(offsetArreglo < VARLEN_STREAM_SIZE /2){
            // tamanio de la clave
            memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
            offsetArreglo += sizeof(int);
            // avanzo la cantidad de caracteres de la clave
            offsetArreglo += sizeof(char) * longClave;
            // avanzo la cantidad de bytes que ocupa la referencia
            offsetArreglo += sizeof(int);

            clavesLeidas++;
        }

        // nuevo offset en el arreglo del nodo actual
        int nuevoOffset = offsetArreglo;

        // copio a partir de ahora todas las claves que restan al nuevo nodo
        for(int i = clavesLeidas; i < _nclaves;i++){
            memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
            offsetArreglo += sizeof(int);

            auxKey = new char[(longClave +1) * sizeof(char)];
            memcpy(auxKey,_pares + offsetArreglo,sizeof(char)*longClave);
            auxKey[longClave] = 0;
            offsetArreglo += sizeof(char)*longClave;

            memcpy(&auxRef,_pares + offsetArreglo, sizeof(int));
            offsetArreglo += sizeof(int);

            if(!clavesCopiadas)
            {
                *claveAlPadre = new char[(longClave + 1) * sizeof(char)];
                strcpy(*claveAlPadre,auxKey);
            }

            ret->insert(auxKey,auxRef);
            delete (auxKey);
            clavesCopiadas++;
        }
```

```
        _offset = nuevoOffset;
        _nclaves = clavesLeidas;
        _espacioLibre = VARLEN_STREAM_SIZE - _offset;
    }

    offsetArreglo = 0;
    // sino la 1º clave del nuevo nodo va a ser la que viene desde el arreglo
    if(!clavesCopiadas)
    {
        // tomo la clave menor para subir al _padre
        memcpy(&longClave,arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        *claveAlPadre = new char[(longClave +1) * sizeof(char)];
        memcpy((*claveAlPadre),       arreglo + offsetArreglo, sizeof(char) * longClave);
        (*claveAlPadre)[longClave] = 0; // coloco la marca de fin de cadena
    }

    //vuelvo a posicionarme en el principio del arreglo para pasar todas las claves
    offsetArreglo = 0;

    // copio las _nclaves de la mitad en adelante en el nuevo nodo
    for(int j = 0; j < clavesArreglo; ++j) {
        // leo la longitud
        memcpy(&longClave,arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        // inicializo la clave y copio su contenido
        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey, arreglo + offsetArreglo, sizeof(char) * longClave);
        offsetArreglo += longClave * sizeof(char);
        auxKey[longClave] = 0; // le coloco la marca de fin de cadena

        memcpy(&auxRef, arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        ret->insert(auxKey, auxRef);
        delete(auxKey);
    }

    // seteo el nuevo hno derecho para el nodo actual
    _hnoDerecho = ret->getId();

    return ret;
}

void VarLenLeafNode::promoteRoot(VarLenNode** nodo1,VarLenNode** nodo2, int id1, int id2, int
clavesArreglo,char** arreglo, int bytesArreglo,
        char** claveARaiz)
{
    // creo los dos nodos que van a ser los hijos de la raiz
    *nodo1 = new VarLenLeafNode(id1, _id, -1, id2);

    *nodo2 = new VarLenLeafNode(id2,_id,id1,-1);

    int offsetArreglo = 0;
    int tamanioClave;
    int auxRef;
    char* auxKey;
```

```
int clavesLeidas = 0;

// copio la mitad de las claves que quedaron en la hoja al nuevo nodo derecho "nodo1"
while(offsetArreglo < VARLEN_STREAM_SIZE / 2){
   //leo el tamaño de la 1º clave
   memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
   offsetArreglo += sizeof(int);

   auxKey = new char[(tamanioClave + 1) * sizeof(char)];
   memcpy(auxKey, _pares + offsetArreglo, sizeof(char) * tamanioClave);
   auxKey[tamanioClave] = 0;
   offsetArreglo += sizeof(char) * tamanioClave;

   memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
   offsetArreglo += sizeof(int);

   (*nodo1)->insert(auxKey, auxRef);

   delete(auxKey);

   clavesLeidas ++;
}

if(clavesLeidas < _nclaves){
   memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
   offsetArreglo += sizeof(int);

   *claveARaiz = new char[(tamanioClave + 1) * sizeof(char)];

   memcpy(*claveARaiz, _pares + offsetArreglo, sizeof(char) * tamanioClave);
   (*claveARaiz)[tamanioClave] = 0;
   offsetArreglo += sizeof(char) * tamanioClave;

   memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
   offsetArreglo += sizeof(int);

   (*nodo2)->insert(*claveARaiz, auxRef);

   clavesLeidas ++;

   for(int i = clavesLeidas;i < _nclaves;i++){
      memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
      offsetArreglo += sizeof(int);

      auxKey = new char[(tamanioClave + 1) * sizeof(char)];
      memcpy(auxKey, _pares + offsetArreglo, sizeof(char) * tamanioClave);
      auxKey[tamanioClave] = 0;
      offsetArreglo += sizeof(char) * tamanioClave;

      memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
      offsetArreglo += sizeof(int);

      (*nodo2)->insert(auxKey, auxRef);

      delete(auxKey);
   }
}
else{
   offsetArreglo = 0;
```

```
        memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        *claveARaiz= new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(*claveARaiz,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
        (*claveARaiz)[tamanioClave] = 0;
    }

    offsetArreglo = 0;

    int clavesCopiadasDesdeArreglo = 0;

    while(clavesCopiadasDesdeArreglo < clavesArreglo){

        memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        auxKey = new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(auxKey, (*arreglo) + offsetArreglo, sizeof(char) * tamanioClave);
        auxKey[tamanioClave] = 0;
        offsetArreglo += sizeof(char) * tamanioClave;

        memcpy(&auxRef, (*arreglo) + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        (*nodo2)->insert(auxKey, auxRef);

        clavesCopiadasDesdeArreglo++;

        delete(auxKey);
    }

    return;
}

void VarLenLeafNode::list(){
    char* auxKey;
    int auxRef;
    int longClave;
    int offset = 0;

    for(int i = 0;i < _nclaves;++i){
        memcpy(&longClave,_pares + offset,sizeof(int));
        offset += sizeof(int);

        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey,_pares + offset, sizeof(char) * longClave);
        auxKey[longClave] = 0;
        offset += sizeof(char) * longClave;

        memcpy(&auxRef,_pares + offset,sizeof(int));
        offset += sizeof(int);

        cout<<"clave: "<<auxKey<<" referencia: "<<auxRef<<endl;

        delete(auxKey);
    }
}
```

**FileDirNode.h**

```
#ifndef FILEDIRNODE_H_INCLUDED
#define FILEDIRNODE_H_INCLUDED

#include <cstdlib>
#include <iostream>
#include <string>

#include "VarLenNode.h"

class FileDirNode : public VarLenNode
{
public:
  // constructors
  FileDirNode(int id = 0, int nivel = 0, int padre = -1);

  virtual int searchFileAndVersion(const char* fileName, int version) = 0;
  virtual int getFirstOf(const char* key) = 0;
};

#endif
```

**FileDirNode.cpp**

```
#include "FileDirNode.h"

FileDirNode::FileDirNode(int id, int nivel, int padre) : VarLenNode(id, nivel, padre) {}
```

**FileDirIndexNode.h**

```
#ifndef NODOBMASINDICE_H_INCLUDED
#define NODOBMASINDICE_H_INCLUDED

#include "FileDirNode.h"

class FileDirIndexNode : public FileDirNode
{
public:
  static const int BYTESREDONDEO;

  // constructors
  FileDirIndexNode(int id = 0, int nivel = 0, int padre = -1, int HijoIzquierdo = 0, char* key = 0, int ref = 0);
        ~FileDirIndexNode();

  void read(char* buffer);
  void write(char* buffer);

  int search(const char* key);
        int searchFileAndVersion(const char* fileName,int version);
  int getFirstOf(const char* key);

  FileDirNode* split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char** claveAlPadre);
  void promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2, int id1, int id2, int clavesArreglo, char** arreglo, int
bytesArreglo, char**  claveARaiz);

  // getters
  int getHijoIzquierdo() const { return _hijoIzquierdo; }
  t_nodeType getType() const { return INDEX; }
```

```
protected:
  int   _hijoIzquierdo;           // la referencia al nodo con claves menores
  char* _redondeo; // bytes de redondeo para completar el tamanio del nodo en disco
};

#endif
```

**FileDirIndexNode.cpp**

```
#include "FileDirIndexNode.h"

#include <string>

const int FileDirIndexNode::BYTESREDONDEO = 2;  // (_hnoIzquierdo + _hnoDerecho) (nodobmashoja) -
_hijoIzquierdo (nodobmasindice)

FileDirIndexNode::FileDirIndexNode(int id, int nivel, int padre, int HijoIzquierdo, char* key, int ref)
                        : FileDirNode(id, nivel, padre)
{
  _hijoIzquierdo = HijoIzquierdo;

  if (BYTESREDONDEO > 0) {
    _redondeo = new char[BYTESREDONDEO + 1];
    strcpy(_redondeo, std::string(BYTESREDONDEO, '*').c_str());
    _redondeo[BYTESREDONDEO] = 0;
  }

  if (ref != 0) {
    insertPair(key, ref);
    _nclaves = 1;
  }
  else _nclaves = 0;
}

FileDirIndexNode::~FileDirIndexNode()
{
  delete _redondeo;
}

void FileDirIndexNode::read(char* buffer)
{
   char* nextByte = buffer;
   readInfoAdm(&nextByte);
   memcpy(&_hijoIzquierdo, nextByte, sizeof(int));
   nextByte += sizeof(int);
   readDatos(&nextByte);

   if (BYTESREDONDEO > 0)
      memcpy(&_redondeo, nextByte, sizeof(char) * BYTESREDONDEO);
}

void FileDirIndexNode::write(char* buffer)
{
   char* nextByte = buffer;
   writeInfoAdm(&nextByte);
   memcpy(nextByte, &_hijoIzquierdo, sizeof(int));
   nextByte += sizeof(int);
   writeDatos(&nextByte);

   if(BYTESREDONDEO > 0)
```

```
        memcpy(nextByte, &_redondeo, sizeof(char) * BYTESREDONDEO);
}

int FileDirIndexNode::search(const char* key)
{
  int ret = _hijoIzquierdo;

  char* auxKey;
        char* auxFileName;
  int auxRef;
        int tamanioClave;
        int offset = 0;

        if(_nclaves){
                int i = 0;
                while (i < _nclaves) {
            memcpy(&tamanioClave,_pares + offset,sizeof(int));
                        offset += sizeof(int);

                        auxKey = new char[(tamanioClave + 1) * sizeof(char)];
                        memcpy(auxKey, _pares + offset,sizeof(char)*tamanioClave);
                        auxKey[tamanioClave] = 0;
                        offset += tamanioClave * sizeof(char);

                        auxFileName = new char[(tamanioClave - 4) * sizeof(char)];
                        memcpy(auxFileName,auxKey,(tamanioClave - 5));
                        auxFileName[tamanioClave - 5] = 0;

                        memcpy(&auxRef, _pares + offset, sizeof(int));
                        offset += sizeof(int);

                        if(strcmp(key, auxFileName) < 0){
                          delete auxKey;
                                delete auxFileName;
                                return ret;
                        }
                        else{
                          ret = auxRef;
                                i++;
                                delete auxKey;
                                delete auxFileName;
                        }
     }

                return ret;
  }

        return ret;
}

int FileDirIndexNode::searchFileAndVersion(const char* fileName,int version)
{
  int ret = _hijoIzquierdo;
  char* auxKey;
        char* auxFileName;
        char* versionStr;
  int auxRef;
        int tamanioClave;
        int offset = 0;
```

```
        if(_nclaves){
          int i = 0;
                while (i < _nclaves) {
                        memcpy(&tamanioClave,_pares + offset,sizeof(int));
                        offset += sizeof(int);

                        auxKey = new char[(tamanioClave + 1) * sizeof(char)];
                        memcpy(auxKey, _pares + offset,sizeof(char)*tamanioClave);
                        auxKey[tamanioClave] = 0;
                        offset += tamanioClave * sizeof(char);

                        auxFileName = new char[(tamanioClave - 4) * sizeof(char)];
                        memcpy(auxFileName,auxKey,(tamanioClave - 5));
                        auxFileName[tamanioClave - 5] = 0;

                        memcpy(&auxRef, _pares + offset, sizeof(int));
                        offset += sizeof(int);

                        if(strcmp(fileName, auxFileName) < 0){
                          delete auxKey;
                                delete auxFileName;
                                return ret;
                        }
                        else if(strcmp(fileName,auxFileName) > 0){
                          ret = auxRef;
                                i++;
                                delete auxKey;
                                delete auxFileName;
                        }
                        else{
                          versionStr = new char[6];
                                memcpy(versionStr,auxKey + (tamanioClave - 5),5*sizeof(char));
                                versionStr[5] = 0;

                                int auxVersion = atoi(versionStr);
                                delete versionStr;
                                delete auxKey;
                                delete auxFileName;

                                if(version >= auxVersion){
                                        ret = auxRef;
                                        i++;
                                }
                                else
                                  return ret;
                        }
                }

          return ret;
  }
        return ret;
}

FileDirNode* FileDirIndexNode::split(int Numero, char* arreglo,int bytesArreglo,int clavesArreglo,
                                                                char** claveAlPadre)
{
        FileDirIndexNode* ret;
  int auxRef;
  char* auxKey;
        int longClave;
```

```
	int offsetArreglo = 0;

	int clavesCopiadas = 0;

	// si tengo 1 sola clave en el arreglo trato de poder sacar alguna del nodo actual
	// para poder insertarla en el nuevo nodo
	if(( clavesArreglo == 1)&&(_espacioLibre < VARLEN_STREAM_SIZE /2)){

		int clavesLeidas = 0;

		while(offsetArreglo < VARLEN_STREAM_SIZE /2){
			// tamanio de la clave
			memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
			offsetArreglo += sizeof(int);
			// avanzo la cantidad de caracteres de la clave
			offsetArreglo += sizeof(char) * longClave;
			// avanzo la cantidad de bytes que ocupa la referencia
			offsetArreglo += sizeof(int);

			clavesLeidas++;
		}

		// nuevo offset en el arreglo del nodo actual
		int nuevoOffset = offsetArreglo;

		// copio a partir de ahora todas las claves que restan al nuevo nodo
		for(int i = clavesLeidas; i < _nclaves;i++){
			memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
			offsetArreglo += sizeof(int);

			auxKey = new char[(longClave +1) * sizeof(char)];
			memcpy(auxKey,_pares + offsetArreglo,sizeof(char)*longClave);
			auxKey[longClave] = 0;
			offsetArreglo += sizeof(char)*longClave;

			memcpy(&auxRef,_pares + offsetArreglo, sizeof(int));
			offsetArreglo += sizeof(int);

			if(!clavesCopiadas)
			{
				*claveAlPadre = new char[(longClave + 1) * sizeof(char)];
				strcpy(*claveAlPadre,auxKey);

				ret = new FileDirIndexNode(Numero,_nivel,_padre,auxRef);
			}
			else
				ret->insert(auxKey,auxRef);

			delete(auxKey);
			clavesCopiadas++;
		}

		_offset = nuevoOffset;
		_nclaves = clavesLeidas;
		_espacioLibre = VARLEN_STREAM_SIZE - _offset;
	}

	offsetArreglo = 0;
	// sino la 1º clave del nuevo nodo va a ser la que viene desde el arreglo
int copiadasDelArreglo = 0;
```

```
            if(!clavesCopiadas){
                        // tomo la clave menor para subir al _padre
                        memcpy(&longClave, arreglo + offsetArreglo, sizeof(int));
                        offsetArreglo += sizeof(int);
                        *claveAlPadre = new char[(longClave + 1)*sizeof(char)];
                        memcpy(*claveAlPadre,arreglo + offsetArreglo,longClave * sizeof(char));
                        (*claveAlPadre)[longClave] = 0;
                        offsetArreglo += longClave * sizeof(char);

                        // tomo la referencia para poder setear el hijo izquierdo
                        memcpy(&auxRef,arreglo + offsetArreglo,sizeof(int));
                        offsetArreglo += sizeof(int);

                        ret = new FileDirIndexNode(Numero,_nivel,_padre,auxRef);
                        copiadasDelArreglo++;
            }

   // copio las claves restantes al nuevo nodo
   for (int j = copiadasDelArreglo; j < clavesArreglo; j++){
      memcpy(&longClave,arreglo + offsetArreglo,sizeof(int));
                        offsetArreglo += sizeof(int);

                        auxKey = new char[(longClave + 1)*sizeof(char)];
                        memcpy(auxKey, arreglo + offsetArreglo, sizeof(char)*longClave);
                        auxKey[longClave] = 0;
                        offsetArreglo += longClave * sizeof(char);

      memcpy(&auxRef, arreglo + offsetArreglo, sizeof(int));
                        offsetArreglo += sizeof(int);

      ret->insert(auxKey,auxRef);
                delete auxKey;
   }

   return ret;
}

void FileDirIndexNode::promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2,int id1,int id2,          int
clavesArreglo,char** arreglo,int bytesArreglo,char** claveARaiz)
{
   int i;
   int auxRef;
   char* auxKey;
            int tamanioClave;
            int offsetArreglo = 0;

   *nodo1 = new FileDirIndexNode(id1,_nivel,_id,_hijoIzquierdo);

            // copio todas las claves de la raiz al nodo1
   for(i = 0; i < _nclaves; ++i) {
                        memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
                        offsetArreglo += sizeof(int);

                        auxKey = new char[(tamanioClave +1) * sizeof(char)];
                        memcpy(auxKey, _pares + offsetArreglo, sizeof(char)*tamanioClave);
                        auxKey[tamanioClave] = 0;
                        offsetArreglo += tamanioClave * sizeof(char);

      memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
```

```
                offsetArreglo += sizeof(int);

  (*nodo1)->insert(auxKey,auxRef);

                delete(auxKey);
}

        // completo hasta tener la mitad del nodo1 "ocupada"
        offsetArreglo = 0;
        int clavesCopiadas = 0;

        while((*nodo1)->getOffset() < VARLEN_STREAM_SIZE / 2)
        {
                memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                auxKey = new char[(tamanioClave +1)*sizeof(char)];
                memcpy(auxKey,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
                auxKey[tamanioClave] = 0;
                offsetArreglo += tamanioClave * sizeof(char);

                memcpy(&auxRef,(*arreglo) + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                (*nodo1)->insert(auxKey,auxRef);

                delete(auxKey);

                clavesCopiadas++;
        }

// tomo la clave que va a ir a la raiz nueva y
        // la referencia que va a ir al hijo izquierdo del nodo 2
        memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        *claveARaiz = new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(*claveARaiz,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
        (*claveARaiz)[tamanioClave] = 0;
        offsetArreglo += sizeof(char)*tamanioClave;

        memcpy(&auxRef,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        clavesCopiadas++;

        // creo el nodo 2 con el hijo izquierdo apuntando a auxRef
*nodo2 = new FileDirIndexNode(id2,_nivel,_id,auxRef);
        //copio las claves y referencias restantes del arreglo
 for (i = clavesCopiadas; i < clavesArreglo; ++i) {
  memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
                offsetArreglo += sizeof(int);

                auxKey = new char[(tamanioClave + 1)*sizeof(char)];
                memcpy(auxKey, (*arreglo) + offsetArreglo, sizeof(char)*tamanioClave);
                auxKey[tamanioClave] = 0;
                offsetArreglo += sizeof(char)*tamanioClave;

  memcpy(&auxRef, (*arreglo) + offsetArreglo, sizeof(int));
                offsetArreglo += sizeof(int);
```

```
    (*nodo2)->insert(auxKey,auxRef);

                delete (auxKey);
                clavesCopiadas++;
    }
}

int FileDirIndexNode::getFirstOf(const char* key)
{
    int ret = _hijoIzquierdo;

    char* auxKey;
        char* auxFileName;
    int auxRef;
        int tamanioClave;
        int offset = 0;

        if(_nclaves){
            int i = 0;
            while (i < _nclaves) {
                memcpy(&tamanioClave,_pares + offset,sizeof(int));
                offset += sizeof(int);

                auxKey = new char[(tamanioClave + 1) * sizeof(char)];
                memcpy(auxKey, _pares + offset,sizeof(char)*tamanioClave);
                auxKey[tamanioClave] = 0;
                offset += tamanioClave * sizeof(char);

                auxFileName = new char[(tamanioClave - 4) * sizeof(char)];
                memcpy(auxFileName,auxKey,(tamanioClave - 5));
                auxFileName[tamanioClave - 5] = 0;

                memcpy(&auxRef, _pares + offset,sizeof(int));
                offset += sizeof(int);

                if(strcmp(key, auxFileName) >= 0){
                    delete auxKey;
                    delete auxFileName;
                    return ret;
                }

                else{
                    ret = auxRef;
                    i++;
                    delete auxKey;
                    delete auxFileName;
                }
            }

            return ret;
        }

        return ret;
}
```

**FileDirLeafNode.h**

```
#ifndef FILEDIRLEAFNODE_H_INCLUDED
#define FILEDIRLEAFNODE_H_INCLUDED

#include "FileDirNode.h"
#include "FileDirIndexNode.h"

class FileDirLeafNode : public FileDirNode
{
public:

   FileDirLeafNode(int id = 0, int padre = 0, int HnoIzquierdo = -1, int HnoDerecho = -1);

   void read (char* buffer);
   void write(char* buffer);

   int search(const char* key);
   int searchFileAndVersion(const char* fileName, int version);
   int getFirstOf(const char* key);

   FileDirNode* split(int Numero, char* arreglo, int bytesArreglo, int clavesArreglo, char** claveAlPadre);
   void promoteRoot(VarLenNode** nodo1, VarLenNode** nodo2, int id1, int id2,int clavesArreglo, char** arreglo, int
bytesArreglo, char** claveARaiz);

   // getters
   int getHnoDerecho()   const { return _hnoDerecho; }
   int getHnoIzquierdo() const { return _hnoIzquierdo; }
   t_nodeType getType() const { return LEAF; }

   // setters
   void setHnoDerecho  (int HnoDerecho) { _hnoDerecho = HnoDerecho; }
   void setHnoIzquierdo(int HnoIzquierdo) { _hnoIzquierdo = HnoIzquierdo; }

   // lista las claves y referencias del nodo
   void list();

private:
   int _hnoDerecho;
   int _hnoIzquierdo;
};

#endif
```

**FileDirLeafNode.cpp**

```
#include "FileDirLeafNode.h"

using std::cout;
using std::endl;

FileDirLeafNode::FileDirLeafNode(int id, int padre, int HnoIzquierdo, int HnoDerecho)
: FileDirNode(id, 0, padre)
{
   _hnoIzquierdo = HnoIzquierdo;
   _hnoDerecho   = HnoDerecho;
}

void FileDirLeafNode::read(char* buffer)
{
```

```
    char* nextByte = buffer;
    readInfoAdm(&nextByte);
    readDatos(&nextByte);
    memcpy(&_hnoIzquierdo, nextByte, sizeof(int));
    nextByte += sizeof(int);
    memcpy(&_hnoDerecho, nextByte, sizeof(int));
}

void FileDirLeafNode::write(char* buffer)
{
    char* nextByte = buffer;
    writeInfoAdm(&nextByte);
    writeDatos(&nextByte);
    memcpy(nextByte, &_hnoIzquierdo, sizeof(int));
    nextByte += sizeof(int);
    memcpy(nextByte, &_hnoDerecho, sizeof(int));
}

int FileDirLeafNode::search(const char* key)
{
    char* auxKey;
    char* auxFileName;
    int longClave;
    int auxRef;
    int i = 0;
    int offsetLectura = 0;
    int ret = -1;

    while (i < _nclaves) {
        memcpy(&longClave,_pares + offsetLectura,sizeof(int));
        offsetLectura += sizeof(int);

        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey, _pares + offsetLectura, sizeof(char) * longClave);
        offsetLectura += sizeof(char) * longClave;
        auxKey[longClave] = 0; // coloco la marca de fin

        // genero el nombre del archivo
        auxFileName = new char[(longClave - 4) * sizeof(char)];
        memcpy(auxFileName,auxKey,(longClave - 5));
        auxFileName[longClave - 5] = 0;

        memcpy(&auxRef, _pares + offsetLectura, sizeof(int));
        offsetLectura += sizeof(int);

        if (strcmp(key, auxFileName) == 0){
            i++;
            ret = auxRef;
            delete auxKey;
            delete auxFileName;
        }
        else if(strcmp(key,auxFileName) > 0)
        {
            i++;
            delete auxKey;
            delete auxFileName;
        }
        else{
            delete auxFileName;
            delete(auxKey);
```

```
            return ret;
        }
    }
    return ret;
}

int FileDirLeafNode::searchFileAndVersion(const char* fileName,int version)
{
    char* auxKey;
    char* auxFileName;
    char* versionStr;
    int auxVersion;
    int longClave;
    int auxRef;
    int i = 0;
    int offsetLectura = 0;
    int ret = -1;

    while (i < _nclaves) {
        memcpy(&longClave,_pares + offsetLectura,sizeof(int));
        offsetLectura += sizeof(int);

        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey, _pares + offsetLectura, sizeof(char) * longClave);
        offsetLectura += sizeof(char) * longClave;
        auxKey[longClave] = 0; // coloco la marca de fin

        // genero el nombre del archivo
        auxFileName = new char[(longClave - 4) * sizeof(char)];
        memcpy(auxFileName,auxKey,(longClave - 5));
        auxFileName[longClave - 5] = 0;

        //genero la clave string
        versionStr = new char[6];
        memcpy(versionStr, auxKey + (longClave - 5),5*sizeof(char));
        versionStr[5] = 0;

        auxVersion = atoi(versionStr);
        delete(versionStr);

        memcpy(&auxRef, _pares + offsetLectura, sizeof(int));
        offsetLectura += sizeof(int);

        if (strcmp(fileName, auxFileName) == 0){
            i++;
            delete auxKey;
            delete auxFileName;

            if(version >= auxVersion)
                ret = auxRef;
            else
                return ret;
        }
        else if(strcmp(fileName,auxFileName) > 0)
        {
            i++;
            delete auxKey;
            delete auxFileName;
        }
        else{
```

```
            delete auxFileName;
            delete(auxKey);
            return ret;
        }
    }
    return ret;
}


FileDirNode* FileDirLeafNode::split(int Numero, char* arreglo,int bytesArreglo,int clavesArreglo, char**
claveAlPadre)
{
    int longClave = 0;
    int offsetArreglo = 0;
    FileDirLeafNode* ret = new FileDirLeafNode(Numero, _padre, _id, _hnoDerecho);
    int auxRef;
    char* auxKey;
    int clavesCopiadas = 0;

    // si tengo 1 sola clave en el arreglo trato de poder sacar alguna del nodo actual
    // para poder insertarla en el nuevo nodo
    if(( clavesArreglo == 1)&&(_espacioLibre < VARLEN_STREAM_SIZE /2)){
        int clavesLeidas = 0;

        while(offsetArreglo < VARLEN_STREAM_SIZE /2){
            // tamanio de la clave
            memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
            offsetArreglo += sizeof(int);
            // avanzo la cantidad de caracteres de la clave
            offsetArreglo += sizeof(char) * longClave;
            // avanzo la cantidad de bytes que ocupa la referencia
            offsetArreglo += sizeof(int);

            clavesLeidas++;
        }

        // nuevo offset en el arreglo del nodo actual
        int nuevoOffset = offsetArreglo;
        // copio a partir de ahora todas las claves que restan al nuevo nodo
        for(int i = clavesLeidas; i < _nclaves;i++){
            memcpy(&longClave,_pares + offsetArreglo,sizeof(int));
            offsetArreglo += sizeof(int);

            auxKey = new char[(longClave +1) * sizeof(char)];
            memcpy(auxKey,_pares + offsetArreglo,sizeof(char)*longClave);
            auxKey[longClave] = 0;
            offsetArreglo += sizeof(char)*longClave;

            memcpy(&auxRef,_pares + offsetArreglo, sizeof(int));
            offsetArreglo += sizeof(int);

            if(!clavesCopiadas)
            {
                *claveAlPadre = new char[(longClave + 1) * sizeof(char)];
                strcpy(*claveAlPadre,auxKey);
            }

            ret->insert(auxKey,auxRef);

            delete (auxKey);
```

```
            clavesCopiadas++;
        }

        _offset = nuevoOffset;
        _nclaves = clavesLeidas;
        _espacioLibre = VARLEN_STREAM_SIZE - _offset;
    }

    offsetArreglo = 0;
    // sino la 1º clave del nuevo nodo va a ser la que viene desde el arreglo
    if(!clavesCopiadas)
    {
        // tomo la clave menor para subir al _padre
        memcpy(&longClave,arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        *claveAlPadre = new char[(longClave +1) * sizeof(char)];
        memcpy((*claveAlPadre), arreglo + offsetArreglo, sizeof(char) * longClave);
        (*claveAlPadre)[longClave] = 0; // coloco la marca de fin de cadena
    }

    //vuelvo a posicionarme en el principio del arreglo para pasar todas las claves
    offsetArreglo = 0;

    // copio las _nclaves de la mitad en adelante en el nuevo nodo
    for(int j = 0; j < clavesArreglo; ++j) {
        // leo la longitud
        memcpy(&longClave,arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        // inicializo la clave y copio su contenido
        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey, arreglo + offsetArreglo, sizeof(char) * longClave);
        offsetArreglo += longClave * sizeof(char);
        auxKey[longClave] = 0; // le coloco la marca de fin de cadena

        memcpy(&auxRef, arreglo + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        ret->insert(auxKey, auxRef);

        delete(auxKey);
    }

    // seteo el nuevo hno derecho para el nodo actual
    _hnoDerecho = ret->getId();

    return ret;
}

void FileDirLeafNode::promoteRoot(VarLenNode** nodo1,VarLenNode** nodo2, int id1, int id2,int
clavesArreglo,char** arreglo,
        int bytesArreglo, char** claveARaiz)
{
    // creo los dos nodos que van a ser los hijos de la raiz
    *nodo1 = new FileDirLeafNode(id1, _id, -1, id2);

    *nodo2 = new FileDirLeafNode(id2,_id,id1,-1);
```

```
int offsetArreglo = 0;
int tamanioClave;
int auxRef;
char* auxKey;
int clavesLeidas = 0;

// copio la mitad de las claves que quedaron en la hoja al nuevo nodo derecho "nodo1"
while(offsetArreglo < VARLEN_STREAM_SIZE / 2){
   //leo el tamaño de la 1º clave
   memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
   offsetArreglo += sizeof(int);

   auxKey = new char[(tamanioClave + 1) * sizeof(char)];
   memcpy(auxKey, _pares + offsetArreglo, sizeof(char) * tamanioClave);
   auxKey[tamanioClave] = 0;
   offsetArreglo += sizeof(char) * tamanioClave;

   memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
   offsetArreglo += sizeof(int);

   (*nodo1)->insert(auxKey, auxRef);

   delete(auxKey);

   clavesLeidas ++;
}

if(clavesLeidas < _nclaves){
   memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
   offsetArreglo += sizeof(int);

   *claveARaiz = new char[(tamanioClave + 1) * sizeof(char)];
   memcpy(*claveARaiz, _pares + offsetArreglo, sizeof(char) * tamanioClave);
   (*claveARaiz)[tamanioClave] = 0;
   offsetArreglo += sizeof(char) * tamanioClave;

   memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
   offsetArreglo += sizeof(int);

   (*nodo2)->insert(*claveARaiz, auxRef);

   clavesLeidas ++;

   for(int i = clavesLeidas;i < _nclaves;i++){
      memcpy(&tamanioClave,_pares + offsetArreglo,sizeof(int));
      offsetArreglo += sizeof(int);

      auxKey = new char[(tamanioClave + 1) * sizeof(char)];

      memcpy(auxKey, _pares + offsetArreglo, sizeof(char) * tamanioClave);
      auxKey[tamanioClave] = 0;
      offsetArreglo += sizeof(char) * tamanioClave;

      memcpy(&auxRef, _pares + offsetArreglo, sizeof(int));
      offsetArreglo += sizeof(int);

      (*nodo2)->insert(auxKey, auxRef);

      delete(auxKey);
   }
```

```
    }
    else{
        offsetArreglo = 0;

        memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        *claveARaiz= new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(*claveARaiz,(*arreglo) + offsetArreglo,sizeof(char)*tamanioClave);
        (*claveARaiz)[tamanioClave] = 0;
    }

    offsetArreglo = 0;

    int clavesCopiadasDesdeArreglo = 0;

    while(clavesCopiadasDesdeArreglo < clavesArreglo){
        memcpy(&tamanioClave,(*arreglo) + offsetArreglo,sizeof(int));
        offsetArreglo += sizeof(int);

        auxKey = new char[(tamanioClave + 1) * sizeof(char)];
        memcpy(auxKey, (*arreglo) + offsetArreglo, sizeof(char) * tamanioClave);
        auxKey[tamanioClave] = 0;
        offsetArreglo += sizeof(char) * tamanioClave;

        memcpy(&auxRef, (*arreglo) + offsetArreglo, sizeof(int));
        offsetArreglo += sizeof(int);

        (*nodo2)->insert(auxKey, auxRef);

        clavesCopiadasDesdeArreglo++;

        delete(auxKey);
    }

    return;
}

void FileDirLeafNode::list(){
    char* auxKey;
    int auxRef;
    int longClave;
    int offset = 0;

    for(int i = 0;i < _nclaves;++i){
        memcpy(&longClave,_pares + offset,sizeof(int));
        offset += sizeof(int);

        auxKey = new char[(longClave + 1) * sizeof(char)];
        memcpy(auxKey,_pares + offset, sizeof(char) * longClave);
        auxKey[longClave] = 0;
        offset += sizeof(char) * longClave;

        memcpy(&auxRef,_pares + offset,sizeof(int));
        offset += sizeof(int);

        cout<<"clave: "<<auxKey<<" referencia: "<<auxRef<<endl;

        delete(auxKey);
    }
```

```
}

int FileDirLeafNode::getFirstOf(const char* key)
{
   char* auxKey;
   char* auxFileName;
   int longClave;
   int auxRef;
   int i = 0;
   int offsetLectura = 0;
   int ret = -1;

   while (i < _nclaves) {
      memcpy(&longClave,_pares + offsetLectura,sizeof(int));
      offsetLectura += sizeof(int);

      auxKey = new char[(longClave + 1) * sizeof(char)];
      memcpy(auxKey, _pares + offsetLectura, sizeof(char) * longClave);
      offsetLectura += sizeof(char) * longClave;
      auxKey[longClave] = 0; // coloco la marca de fin

      // genero el nombre del archivo
      auxFileName = new char[(longClave - 4) * sizeof(char)];
      memcpy(auxFileName,auxKey,(longClave - 5));
      auxFileName[longClave - 5] = 0;

      memcpy(&auxRef, _pares + offsetLectura, sizeof(int));
      offsetLectura += sizeof(int);

      if (strcmp(key, auxFileName) >=  0){
         i++;
         ret = auxRef;
         delete auxKey;
         delete auxFileName;
      }
      else{
         delete auxFileName;
         delete(auxKey);
         return ret;
      }
   }
   return ret;
}
```

**bplustree.h**

```
#ifndef BPLUSTREE_H_INCLUDED
#define BPLUSTREE_H_INCLUDED

#include <fstream>
#include <string>

#include "Node.h"

using std::string;

class BPlusTree
{
public:
  BPlusTree();
  virtual ~BPlusTree();

  bool create(const string& a_Filename);
  bool destroy();

  virtual bool open(const string& a_Filename) = 0;
  virtual bool close() = 0;

  virtual int search(const char* key) = 0;
  virtual bool insert(const char* key, int reference) = 0;

protected:
  virtual bool readHeader() = 0;
  virtual bool writeHeader() = 0;

  virtual bool readRoot() = 0;
  virtual bool writeRoot() = 0;

  virtual bool readNode(int id, Node** node) = 0;
  virtual bool writeNode(Node* node) = 0;

  virtual int searchPlace(const char* key) = 0;

  virtual int searchPlaceRec(const char* key) = 0;

  virtual void insertarEnPadre(int NroNodoPadre, int NroNodoHijo, const char* claveAlPadre) = 0;

  virtual bool isEmpty() const = 0;

  int           _nNodos;
  char*         _buffer;
  std::fstream  _filestr;
  string        _filename;
  bool          _isOpen;
};

#endif
```

**bplustree.cpp**

```
#include "bplustree.h"
#include "debug.h"

using std::ios;

BPlusTree::BPlusTree() : _nNodos(0), _isOpen(false)
{
}

BPlusTree::~BPlusTree()
{
  delete _buffer;
}

bool BPlusTree::create(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("creating bplustree in " + a_Filename + "\n");
   _filestr.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

          if (!_filestr.is_open()) {
                    _filestr.open(a_Filename.c_str(), ios::out | ios::binary);
                    _filestr.close();

                    _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
                    if (!_filestr.is_open()) {
         _isOpen = false;
         debug("bplustree creation failed\n");
                              return false;
      }
           }

   _nNodos = 0;
   _isOpen = writeHeader();
   debug("bplustree creation " + string(_isOpen ? "successfull" : "failed") + "\n");

   return _isOpen;
}

bool BPlusTree::destroy()
{
   if (_isOpen)
      return false;

   debug("destroying bplustree '" + _filename + "'\n");
   int ret = remove(_filename.c_str());
   debug("arbolbmas bplustree " + string((ret == 0) ? "successfull" : "failed") + "\n");
   return (ret == 0);
}
```

**FixLenBPlusTree.h**

```
#ifndef FIXLENBPLUSTREE_H_INCLUDED
#define FIXLENBPLUSTREE_H_INCLUDED

#include "bplustree.h"
#include "FixLenNode.h"
#include "FixLenIndexNode.h"
#include "FixLenLeafNode.h"

class FixLenBPlusTree : public BPlusTree
{
public:
  FixLenBPlusTree();
  virtual ~FixLenBPlusTree();

  bool open(const string& a_Filename);
  bool close();

  int search(const char* key);

  bool insert(const char* key,int reference);

protected:
  bool readHeader();
  bool writeHeader();

  bool readRoot();
  bool writeRoot();

  bool readNode(int id, Node** node);
  bool writeNode(Node* node);

  int searchPlace(const char* key);

  int searchPlaceRec(const char* key);

  void insertarEnPadre(int NroNodoPadre, int NroNodoHijo, const char* claveAlPadre);

  bool actualizarPadre(FixLenIndexNode* padre);

  bool isEmpty() const { return _raiz == 0; }

private:
  FixLenNode* _raiz;
  FixLenNode* _nodoActual;
};

#endif
```

**FixLenBPlusTree.cpp**

```
#include "FixLenBPlusTree.h"
#include "debug.h"

using std::ios;

FixLenBPlusTree::FixLenBPlusTree() : BPlusTree()
{
  _raiz = 0;
```

```
  _nodoActual = 0;
  _buffer = new char[FixLenNode::FIXLEN_NODE_SIZE];
}

FixLenBPlusTree::~FixLenBPlusTree()
{
  delete _buffer;
  delete _nodoActual;
  delete _raiz;
}

bool FixLenBPlusTree::open(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("opening bplustree '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
   _isOpen = _filestr.is_open() && readHeader();

   if (_nNodos > 0)
      _isOpen = _isOpen && readRoot();

   if (_isOpen) {
      _nodoActual = _raiz;
      _filename = a_Filename;
   }
   debug("bplustree open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool FixLenBPlusTree::close()
{
   if (!_isOpen)
      return true;

   bool ret = true;
   debug("closing bplustree " + _filename + "\n");
   if (writeHeader()) {
      if((_raiz != 0)  && writeRoot())
       if (_raiz != _nodoActual)
          ret = writeNode(_nodoActual);
   }
   else
      ret = false;

   _filestr.close();
   _isOpen = _filestr.is_open();
   debug("bplustree close " + string((!_isOpen && ret) ? "successfull" : "failed") + "\n");
   return (!_isOpen && ret);
}

bool FixLenBPlusTree::insert(const char* key,int reference)
{
            FixLenLeafNode* hojaNueva;
   if (_nNodos) {
      int nodoReceptor = searchPlace(key);
      if (nodoReceptor) {              // voy a insertar en un nodo != _raiz
         switch (_nodoActual->insert(key, reference)) {
         case 1:
```

```
            return true;
                                        break;

        case 2:
            int nroNuevoNodo;

            // creo una nueva hoja para insertar la clave que no entra por overflow
            hojaNueva = new FixLenLeafNode(_nNodos, _nodoActual->getPadre(),_nodoActual->getId(), -1);

                                        // le inserto la clave que me sobro
            hojaNueva->insert(key, reference);

                                        (static_cast<FixLenLeafNode*>(_nodoActual))->setHnoDerecho(hojaNueva-
>getId());

            // incremento la cantidad de nodos
            _nNodos++;

            nroNuevoNodo = hojaNueva->getId();

            // escribo los dos nodos
            writeNode(_nodoActual);
            writeNode(hojaNueva);

            delete hojaNueva;

            insertarEnPadre(_nodoActual->getPadre(), nroNuevoNodo, key);
            return true;
                                        break;

        default:
            return false;
                                        break;
        }
      }
      else {
        FixLenIndexNode* nuevaRaiz;
        int idH1;
        int idH2;
        FixLenLeafNode* nuevaHoja;
        FixLenLeafNode* nuevaHoja2;
        switch (_raiz->insert(key, reference)) {
        case 1:
            return true;

        case 2:
            nuevaHoja = 0;
            nuevaHoja2 = 0;

            _raiz->promoteRoot((FixLenNode**)(&nuevaHoja),_nNodos);
                                        nuevaHoja2 = new FixLenLeafNode(_nNodos + 1, _raiz->getId(), nuevaHoja-
>getId(), -1);
                                        nuevaHoja->setHnoDerecho(nuevaHoja2->getId());
            // inserto la clave que me genero el overflow en la nueva hoja
            nuevaHoja2->insert(key,reference);

            _nNodos += 2;

            // escribo las 2 nuevas hojas
            writeNode(nuevaHoja);
```

```
            writeNode(nuevaHoja2);

            idH1 = nuevaHoja->getId();
            idH2 = nuevaHoja2->getId();

            nuevaRaiz = (static_cast<FixLenLeafNode*>(_raiz))->convertirAIndice(idH1,idH2,key);

            delete(_raiz);

            _nodoActual = _raiz = nuevaRaiz;

            return true;

          default:
            return false;
          }
        }
    }

    else {
        _raiz = new FixLenLeafNode();
        _nNodos++;
        _nodoActual = _raiz;
        _raiz->insert(key, reference);
    }
    return true;
}

bool FixLenBPlusTree::readHeader()
{
    if (_filestr.is_open()) {
        char* nextByte = _buffer;
        _filestr.seekg(0, ios::beg);
        _filestr.read(_buffer, FixLenNode::FIXLEN_NODE_SIZE);
        // leo la cantidad de nodos
        memcpy(&_nNodos, nextByte, sizeof(int));
        return true;
    }
    return false;
}

bool FixLenBPlusTree::writeHeader()
{
    if (_filestr.is_open()) {
        // clear buffer
        memset(_buffer,'\0', FixLenNode::FIXLEN_NODE_SIZE);
        char* nextByte = _buffer;
        // volcar en _buffer la cantidad de nodos
        memcpy(nextByte, &_nNodos, sizeof(int));
        // volcar al archivo
        _filestr.seekg(0, ios::beg);
        _filestr.seekp(0, ios::beg);
        _filestr.write(_buffer, FixLenNode::FIXLEN_NODE_SIZE);
        return true;
    }
    return false;
}

bool FixLenBPlusTree::readRoot()
{
```

```
    if (_filestr.is_open()) {
        _filestr.seekg(FixLenNode::FIXLEN_NODE_SIZE, ios::beg);
        _filestr.read(_buffer, FixLenNode::FIXLEN_NODE_SIZE);

        if (_nNodos > 1)
            _raiz = new FixLenIndexNode();
        else
            _raiz = new FixLenLeafNode();

        _raiz->read(_buffer);
        return true;
    }
    return false;
}

bool FixLenBPlusTree::writeRoot()
{
    if (_filestr.is_open()) {
        if (_raiz != 0)
            _raiz->write(_buffer);

        if (_filestr.fail())
            _filestr.clear();

        _filestr.seekg(FixLenNode::FIXLEN_NODE_SIZE, ios::beg);
        _filestr.seekp(FixLenNode::FIXLEN_NODE_SIZE, ios::beg);
        _filestr.write(_buffer, FixLenNode::FIXLEN_NODE_SIZE);
        return true;
    }
    return false;
}

bool FixLenBPlusTree::readNode(int id, Node** node)
{
    if (_filestr.is_open()) {
        if ((*node != _raiz) && (*node != 0)) {
            writeNode(*node);
            delete (*node);
        }
        else
            *node = 0;

        _filestr.seekg(FixLenNode::FIXLEN_NODE_SIZE * (id + 1), ios::beg);
        _filestr.read(_buffer, FixLenNode::FIXLEN_NODE_SIZE);

        int nivelNodo;
        memcpy(&nivelNodo, _buffer, sizeof(int));

        if (nivelNodo != 0)
            (*node) = new FixLenIndexNode();
        else
            (*node) = new FixLenLeafNode();

        (*node)->read(_buffer);
        return true;
    }
    return false;
}

bool FixLenBPlusTree::writeNode(Node* node)
```

```
{
   if (_filestr.is_open()) {
      node->write(_buffer);

      if(_filestr.fail())
         _filestr.clear();

      _filestr.seekp(FixLenNode::FIXLEN_NODE_SIZE * (node->getId() + 1), ios::beg);

      if (_filestr.fail())
         _filestr.clear();

      _filestr.write(_buffer, FixLenNode::FIXLEN_NODE_SIZE);
      return true;
   }
   return false;
}

void FixLenBPlusTree::insertarEnPadre(int idNodoPadre, int idNodoHijo,const char* claveAlPadre)
{
  FixLenIndexNode* nuevoIndice = 0;
  FixLenIndexNode* nuevoIndice2 = 0;
           FixLenIndexNode* nuevaRaiz = 0;

   if (idNodoPadre != 0) { // el padre es distinto de la raiz
      readNode(idNodoPadre, (Node**)&_nodoActual);

      if (_nodoActual->insert(claveAlPadre, idNodoHijo) == 2) {
         nuevoIndice = new FixLenIndexNode(_nNodos, _nodoActual->getNivel(),
                                                                                                  _nodoActual-
>getPadre(), idNodoHijo);

         _nNodos++;
         // escribo los nodos en disco
         writeNode(_nodoActual);
         writeNode(nuevoIndice);

         // actualizo el puntero al padre en los hijos del nuevo nodo indice
         actualizarPadre(nuevoIndice);

         insertarEnPadre(nuevoIndice->getPadre(), nuevoIndice->getId(),claveAlPadre);

                              delete nuevoIndice;

         return;
      }
      return;
   }

   // si llego hasta aca es porque el que esta "rebalsado" es la raiz

   if (_raiz->insert(claveAlPadre, idNodoHijo) == 2) {

      _raiz->promoteRoot( (FixLenNode**)(&nuevoIndice),
                  _nNodos);

      nuevoIndice2 = new FixLenIndexNode(_nNodos + 1, _raiz->getNivel(), _raiz->getId(), idNodoHijo);

      nuevaRaiz = new FixLenIndexNode(_raiz->getId(), _raiz->getNivel() + 1,
                                                      _raiz->getPadre(), nuevoIndice->getId(), claveAlPadre,
```

```
nuevoIndice2->getId());

                        if(_nodoActual != _raiz) {
                                delete _raiz;
                                _raiz = nuevaRaiz;
                        }

                        else {
                                delete _raiz;
                                _raiz = _nodoActual = nuevaRaiz;
                        }

        writeNode(nuevoIndice);
        writeNode(nuevoIndice2);

        _nNodos += 2;

        actualizarPadre(nuevoIndice);
        actualizarPadre(nuevoIndice2);

        if (nuevoIndice) delete nuevoIndice;
        if (nuevoIndice2) delete nuevoIndice2;
    }
}

bool FixLenBPlusTree::actualizarPadre(FixLenIndexNode* padre)
{
    // le seteo el padre al hijo izquierdo
    readNode(padre->getHijoIzquierdo(), (Node**)&_nodoActual);
    _nodoActual->setPadre(padre->getId());

    for(int i = 0; i < padre->getCantClaves(); ++i) {
        readNode(padre->getRef(i), (Node**)&_nodoActual);
        _nodoActual->setPadre(padre->getId());
    }

    return writeNode(_nodoActual);
}

int FixLenBPlusTree::searchPlace(const char* key)
{
    if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
        writeNode(_nodoActual);
        delete _nodoActual;
        _nodoActual = _raiz;
    }

    return searchPlaceRec(key);
}

int FixLenBPlusTree::searchPlaceRec(const char* key)
{
    if (_nodoActual->getType() == FixLenNode::LEAF)
        return _nodoActual->getId();
    else {
        int indice = _nodoActual->search(key);

        if (readNode(indice, (Node**)&_nodoActual))
            return searchPlaceRec(key);
```

```
        return -1;
    }
}

int FixLenBPlusTree::search(const char* key)
{
    if (isEmpty())
        return -1;

    if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
        writeNode(_nodoActual);
        delete _nodoActual;
        _nodoActual = _raiz;
    }

    while (_nodoActual->getType() != FixLenNode::LEAF) {
        int proximoALeer = _nodoActual->search(key);
        readNode(proximoALeer, (Node**)&_nodoActual);
    }

    return _nodoActual->search(key);
}
```

**VarLenBPlusTree.h**

```
#ifndef VARLENBPLUSTREE_H_INCLUDED
#define VARLENBPLUSTREE_H_INCLUDED

#include "bplustree.h"
#include "VarLenNode.h"
#include "VarLenIndexNode.h"
#include "VarLenLeafNode.h"
#include <fstream>
#include <string>

using std::string;

class VarLenBPlusTree : public BPlusTree
{
public:
  VarLenBPlusTree();
  virtual ~VarLenBPlusTree();

  bool open(const string& a_Filename);
  bool close();

  bool insert(const char* key, int reference);

  int search(const char* key);

protected:
  bool readHeader();
  bool writeHeader();

  bool readRoot();
  bool writeRoot();

  bool readNode(int id, Node** node);
  bool writeNode(Node* node);

  int searchPlace(const char* key);

  int searchPlaceRec(const char* key);

  void insertarEnPadre(int NroNodoPadre, int NroNodoHijo,const char* claveAlPadre);

  bool actualizarPadre(VarLenIndexNode* padre);

  bool isEmpty() const { return _raiz == 0; }

private:
  VarLenNode* _raiz;
  VarLenNode* _nodoActual;
};

#endif
```

**VarLenBPlusTree.cpp**

```
#include "VarLenBPlusTree.h"
#include "debug.h"

using std::ios;

VarLenBPlusTree::VarLenBPlusTree() : BPlusTree()
{
  _raiz = 0;
  _nodoActual = 0;
  _buffer = new char[VarLenNode::VARLEN_NODE_SIZE];
}

VarLenBPlusTree::~VarLenBPlusTree()
{
  delete _buffer;
  delete _raiz;
  delete _nodoActual;
}

bool VarLenBPlusTree::open(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("opening bplustree '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
   _isOpen = _filestr.is_open() && readHeader();

   if (_nNodos > 0)
      _isOpen = _isOpen && readRoot();

   if (_isOpen) {
      _nodoActual = _raiz;
      _filename = a_Filename;
   }
   debug("bplustree open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool VarLenBPlusTree::close()
{
   if (!_isOpen)
      return true;

   bool ret = true;
   debug("closing bplustree " + _filename + "\n");
   if (writeHeader()) {
      if((_raiz != 0)  && writeRoot())
       if (_raiz != _nodoActual)
          ret = writeNode(_nodoActual);
   }
   else
      ret = false;

   _filestr.close();
   _isOpen = _filestr.is_open();
   debug("bplustree close " + string((!_isOpen && ret) ? "successfull" : "failed") + "\n");
   return (!_isOpen && ret);
```

```
}

bool VarLenBPlusTree::insert(const char* key, int reference)
{
   VarLenLeafNode* hojaNueva;
   VarLenLeafNode* hnoDerecho;
            int bytesArreglo = 0;
            int clavesArreglo = 0;
            char* arreglo;
            char* claveARaiz;

   if (_nNodos > 0) {
      int nodoReceptor = searchPlace(key);
      if (nodoReceptor) {              // voy a insertar en un nodo != _raiz
         char* claveAlPadre;
         switch (_nodoActual->insert(key, reference, &clavesArreglo, &arreglo, &bytesArreglo)) {
         case 1:
            return true;

         case 2:
            int nroNuevoNodo;

            // creo una nueva hoja haciendo el split
            hojaNueva = static_cast<VarLenLeafNode*>((static_cast<VarLenLeafNode*> (_nodoActual))->split

           (_nNodos,arreglo,bytesArreglo,clavesArreglo,&claveAlPadre));

                                          // libero el arreglo
                                          delete(arreglo);

            // tengo que levantar el hno derecho de la nueva hoja y setearle la nueva hoja
            // como su hno izquierdo
            if (hojaNueva->getHnoDerecho() > 0) {
               hnoDerecho = 0;
               readNode(hojaNueva->getHnoDerecho(), (Node**)&hnoDerecho);
               hnoDerecho->setHnoIzquierdo(hojaNueva->getId());
               writeNode(hnoDerecho);
               delete hnoDerecho;
            }

            // incremento la cantidad de nodos
            _nNodos++;

            nroNuevoNodo = hojaNueva->getId();

            // escribo los dos nodos
            writeNode(_nodoActual);
            writeNode(hojaNueva);

            delete hojaNueva;
            insertarEnPadre(_nodoActual->getPadre(), nroNuevoNodo, claveAlPadre);
                                          delete(claveAlPadre);
            return true;

         default:
            return false;
         }
      }
      else {
         VarLenIndexNode* nuevaRaiz;
```

```
        int idH1;
        int idH2;
        VarLenLeafNode* nuevaHoja;
        VarLenLeafNode* nuevaHoja2;
        switch (_raiz->insert(key,reference, &clavesArreglo, &arreglo, &bytesArreglo)) {
        case 1:
           return true;

        case 2:
           nuevaHoja = 0;
           nuevaHoja2 = 0;

           // creo 2 nodos hoja nuevos para hacer el split y hacer que la nueva raiz sea indice
           _raiz->promoteRoot( (VarLenNode**)(&nuevaHoja),
                        (VarLenNode**)(&nuevaHoja2),
                        _nNodos, _nNodos + 1,
                                                                    clavesArreglo, &arreglo, bytesArreglo,
&claveARaiz);

           _nNodos += 2;

           // escribo las 2 nuevas hojas
           writeNode(nuevaHoja);
           writeNode(nuevaHoja2);

           idH1 = nuevaHoja->getId();
           idH2 = nuevaHoja2->getId();

           nuevaRaiz = new VarLenIndexNode(_raiz->getId(), _raiz->getNivel() + 1, _raiz->getPadre(),
                                                                        idH1, claveARaiz,
idH2);

           delete _raiz;
           _nodoActual = _raiz = nuevaRaiz;
           return true;

        default:
           return false;
        }
                     }
   }

   else {
      _raiz = new VarLenLeafNode();
      _nNodos++;
      _nodoActual = _raiz;
      _raiz->insert(key, reference);
   }
   return true;
}

bool VarLenBPlusTree::readHeader()
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      _filestr.seekg(0, ios::beg);
      _filestr.read(_buffer, VarLenNode::VARLEN_NODE_SIZE);
      // leo la cantidad de nodos
      memcpy(&_nNodos, nextByte, sizeof(int));
      return true;
```

```
    }
    return false;
}

bool VarLenBPlusTree::writeHeader()
{
    if (_filestr.is_open()) {
        // clear buffer
        memset(_buffer,'\0', VarLenNode::VARLEN_NODE_SIZE);
        char* nextByte = _buffer;
        // volcar en _buffer la cantidad de nodos
        memcpy(nextByte, &_nNodos, sizeof(int));
        // volcar al archivo
        _filestr.seekg(0, ios::beg);
        _filestr.seekp(0, ios::beg);
        _filestr.write(_buffer, VarLenNode::VARLEN_NODE_SIZE);
        return true;
    }
    return false;
}

bool VarLenBPlusTree::readRoot()
{
    if (_filestr.is_open()) {
        _filestr.seekg(VarLenNode::VARLEN_NODE_SIZE, ios::beg);
        _filestr.read(_buffer, VarLenNode::VARLEN_NODE_SIZE);

        if (_nNodos > 1)
            _raiz = new VarLenIndexNode();
        else
            _raiz = new VarLenLeafNode();

        _raiz->read(_buffer);
        return true;
    }
    return false;
}

bool VarLenBPlusTree::writeRoot()
{
    if (_filestr.is_open()) {
        if (_raiz != 0)
            _raiz->write(_buffer);

        if (_filestr.fail())
            _filestr.clear();

        _filestr.seekg(VarLenNode::VARLEN_NODE_SIZE, ios::beg);
        _filestr.seekp(VarLenNode::VARLEN_NODE_SIZE, ios::beg);
        _filestr.write(_buffer, VarLenNode::VARLEN_NODE_SIZE);
        return true;
    }
    return false;
}

bool VarLenBPlusTree::readNode(int id, Node** node)
{
    if (_filestr.is_open()) {
        if ((*node != _raiz) && (*node != 0)) {
            writeNode(*node);
```

```
            delete (*node);
        }
        else
            *node = 0;

        _filestr.seekg(VarLenNode::VARLEN_NODE_SIZE * (id + 1), ios::beg);
        _filestr.read(_buffer, VarLenNode::VARLEN_NODE_SIZE);

        int nivelNodo;
        memcpy(&nivelNodo, _buffer, sizeof(int));

        if (nivelNodo != 0)
            (*node) = new VarLenIndexNode();
        else
            (*node) = new VarLenLeafNode();

        (*node)->read(_buffer);
        return true;
    }
    return false;
}

bool VarLenBPlusTree::writeNode(Node* node)
{
    if (_filestr.is_open()) {
        node->write(_buffer);

        if(_filestr.fail())
            _filestr.clear();

        _filestr.seekp(VarLenNode::VARLEN_NODE_SIZE * (node->getId() + 1), ios::beg);

        if (_filestr.fail())
            _filestr.clear();

        _filestr.write(_buffer, VarLenNode::VARLEN_NODE_SIZE);
        return true;
    }
    return false;
}

void VarLenBPlusTree::insertarEnPadre(int NroNodoPadre, int NroNodoHijo,const char* claveAlPadre)
{
    VarLenIndexNode* nuevoIndice = 0;
    VarLenIndexNode* nuevoIndice2 = 0;
            char* arreglo;
            int bytesArreglo;
            int clavesArreglo;
            char* claveASubir;
            char* claveARaiz;

    if (NroNodoPadre != 0) { // el padre es distinto de la raiz
        readNode(NroNodoPadre, (Node**)&_nodoActual);

        if (_nodoActual->insert(claveAlPadre, NroNodoHijo, &clavesArreglo, &arreglo, &bytesArreglo) == 2) {
            // hago el split
            nuevoIndice = static_cast<VarLenIndexNode*>((static_cast<VarLenIndexNode*>(_nodoActual))->split
                                                                    (_nNodos, arreglo, bytesArreglo,
clavesArreglo, &claveASubir));
```

```
                                delete(arreglo);
            _nNodos++;
            // escribo los nodos en disco
            writeNode(_nodoActual);
            writeNode(nuevoIndice);

            // actualizo el puntero al padre en los hijos del nuevo nodo indice
            actualizarPadre(nuevoIndice);

            int referenciaNuevoNodo = nuevoIndice->getId();

                                int padreNuevo = nuevoIndice->getPadre();

            delete nuevoIndice;

            insertarEnPadre(padreNuevo, referenciaNuevoNodo, claveASubir);

                                delete(claveASubir);
            return;
        }
        return;
    }

    // si llego hasta aca es porque el que esta "rebalsado" es la raiz

    if (_raiz->insert(claveAlPadre, NroNodoHijo, &clavesArreglo, &arreglo, &bytesArreglo) == 2) {

        _raiz->promoteRoot( (VarLenNode**)(&nuevoIndice),
                        (VarLenNode**)(&nuevoIndice2),
                        _nNodos,_nNodos + 1,
                                                                clavesArreglo,&arreglo,bytesArreglo,&claveARaiz);

        delete(arreglo);

        writeNode(nuevoIndice);
        writeNode(nuevoIndice2);

        _nNodos += 2;

        actualizarPadre(nuevoIndice);
        actualizarPadre(nuevoIndice2);

                        VarLenIndexNode* nuevaRaiz = new VarLenIndexNode(_raiz->getId(),_raiz->getNivel() + 1,_raiz-
>getPadre(),

              nuevoIndice->getId(),claveARaiz,nuevoIndice2->getId());

        if (nuevoIndice) delete nuevoIndice;
        if (nuevoIndice2) delete nuevoIndice2;

                        _raiz = nuevaRaiz;

                        delete(claveARaiz);

    }
}

bool VarLenBPlusTree::actualizarPadre(VarLenIndexNode* padre)
{
    // le seteo el padre al hijo izquierdo
```

```
    readNode(padre->getHijoIzquierdo(), (Node**)&_nodoActual);
    _nodoActual->setPadre(padre->getId());

    for(int i = 1; i <= padre->getCantClaves(); ++i) {
        readNode(padre->getRef(i), (Node**)&_nodoActual);
        _nodoActual->setPadre(padre->getId());
    }

    return writeNode(_nodoActual);
}

int VarLenBPlusTree::searchPlace(const char* key)
{
    if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
        writeNode(_nodoActual);
        delete _nodoActual;
        _nodoActual = _raiz;
    }

    return searchPlaceRec(key);
}

int VarLenBPlusTree::searchPlaceRec(const char* key)
{
    if (_nodoActual->getType() == VarLenNode::LEAF)
        return _nodoActual->getId();
    else {
        int indice = _nodoActual->search(key);

        if (readNode(indice, (Node**)&_nodoActual))
            return searchPlaceRec(key);

        return -1;
    }
}

int VarLenBPlusTree::search(const char* key)
{
    if (isEmpty())
        return -1;

    if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
        writeNode(_nodoActual);
        delete _nodoActual;
        _nodoActual = _raiz;
    }

    while (_nodoActual->getType() != VarLenNode::LEAF) {
        int proximoALeer = _nodoActual->search(key);
        readNode(proximoALeer, (Node**)&_nodoActual);
    }

    return _nodoActual->search(key);
}
```

**FileDirBPlusTree.h**

```
#ifndef FILEDIRBPLUSTREE_H_INCLUDED
#define FILEDIRBPLUSTREE_H_INCLUDED

#include "bplustree.h"
#include "FileDirNode.h"
#include "FileDirIndexNode.h"
#include "FileDirLeafNode.h"
#include <fstream>
#include <string>

using std::string;

class FileDirBPlusTree : public BPlusTree
{
public:
  FileDirBPlusTree();
  virtual ~FileDirBPlusTree();

  bool open(const string& a_Filename);
  bool close();

  int search(const char* key);
  int searchFileAndVersion(const char* fileName, int version);
  int getFirstBlock(const char* key);

  bool insert(const char* key, int reference);

  void list();

protected:

  bool readHeader();
  bool writeHeader();

  bool readRoot();
  bool writeRoot();

  bool readNode(int id,Node** node);
  bool writeNode(Node* node);

  int searchPlace(const char* key);

  int searchPlaceRec(const char* key);

  void insertarEnPadre(int NroNodoPadre, int NroNodoHijo,const char* claveAlPadre);

  bool actualizarPadre(FileDirIndexNode* padre);

  bool isEmpty() const { return _raiz == 0; }

private:
  FileDirNode*   _raiz;
  FileDirNode*   _nodoActual;
};
#endif
```

**FileDirBPlusTree.cpp**

```
#include "FileDirBPlusTree.h"
#include "debug.h"

using std::ios;

FileDirBPlusTree::FileDirBPlusTree() : BPlusTree()
{
   _raiz = 0;
   _nodoActual = 0;
   _buffer = new char[FileDirNode::VARLEN_NODE_SIZE];
}

FileDirBPlusTree::~FileDirBPlusTree()
{
  delete _buffer;
  delete _raiz;
  delete _nodoActual;
}

bool FileDirBPlusTree::open(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("opening bplustree '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
   _isOpen = _filestr.is_open() && readHeader();

   if (_nNodos > 0)
      _isOpen = _isOpen && readRoot();

   if (_isOpen) {
      _nodoActual = _raiz;
      _filename = a_Filename;
   }
   debug("bplustree open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool FileDirBPlusTree::close()
{
   if (!_isOpen)
      return true;

   bool ret = true;
   debug("closing bplustree " + _filename + "\n");
   if (writeHeader()) {
      if((_raiz != 0)  && writeRoot())
       if (_raiz != _nodoActual)
          ret = writeNode(_nodoActual);
   }
   else
      ret = false;

   _filestr.close();
   _isOpen = _filestr.is_open();
   debug("bplustree close " + string((!_isOpen && ret) ? "successfull" : "failed") + "\n");
   return (!_isOpen && ret);
```

```
}

int FileDirBPlusTree::search(const char*key)
{
   if (isEmpty())
      return -1;

   if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
      writeNode(_nodoActual);
      delete _nodoActual;
      _nodoActual = _raiz;
   }

   while (_nodoActual->getType() != FileDirNode::LEAF) {
      int proximoALeer = _nodoActual->search(key);
      readNode(proximoALeer, (Node**)&_nodoActual);
   }

   return _nodoActual->search(key);
}

int FileDirBPlusTree::searchFileAndVersion(const char* fileName, int version)
{
   if (isEmpty())
      return -1;

   if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
      writeNode(_nodoActual);
      delete _nodoActual;
      _nodoActual = _raiz;
   }

   while (_nodoActual->getType() != FileDirNode::LEAF) {
      int proximoALeer = _nodoActual->searchFileAndVersion(fileName, version);
      readNode(proximoALeer,(Node**) &_nodoActual);
   }

         return _nodoActual->searchFileAndVersion(fileName, version);
}

int FileDirBPlusTree::getFirstBlock(const char* key)
{
   if(isEmpty())
     return -1;

   if((_nodoActual != _raiz) && (_nodoActual != 0)){
      writeNode(_nodoActual);
      delete _nodoActual;
      _nodoActual = _raiz;
   }

   while(_nodoActual->getType() != FileDirNode::LEAF){
      int proximoALeer = (static_cast<FileDirIndexNode*>(_nodoActual)->getFirstOf(key));
      readNode(proximoALeer, (Node**)&_nodoActual);
   }

   return _nodoActual->getFirstOf(key);
}

bool FileDirBPlusTree::insert(const char* key, int reference)
```

```
{
  FileDirLeafNode* hojaNueva;
  FileDirLeafNode* hnoDerecho;
            int bytesArreglo = 0;
            int clavesArreglo = 0;
            char* arreglo;
            char* claveARaiz;

  if (_nNodos > 0) {
    int nodoReceptor = searchPlace(key);
    if (nodoReceptor) {              // voy a insertar en un nodo != _raiz
        char* claveAlPadre;
        switch (_nodoActual->insert(key, reference, &clavesArreglo, &arreglo, &bytesArreglo)) {
        case 1:
           return true;
        case 2:
           int nroNuevoNodo;
           // creo una nueva hoja haciendo el split
           hojaNueva = static_cast<FileDirLeafNode*>((static_cast<FileDirLeafNode*> (_nodoActual))->split

          (_nNodos,arreglo,bytesArreglo,clavesArreglo,&claveAlPadre));
                                          // libero el arreglo
                                          delete(arreglo);

           // tengo que levantar el hno derecho de la nueva hoja y setearle la nueva hoja
           // como su hno izquierdo
           if (hojaNueva->getHnoDerecho() > 0) {
              hnoDerecho = 0;
              readNode(hojaNueva->getHnoDerecho(), (Node**)&hnoDerecho);
              hnoDerecho->setHnoIzquierdo(hojaNueva->getId());
              writeNode(hnoDerecho);
              delete hnoDerecho;
           }

           // incremento la cantidad de nodos
           _nNodos++;
           nroNuevoNodo = hojaNueva->getId();
           // escribo los dos nodos
           writeNode(_nodoActual);
           writeNode(hojaNueva);

           delete hojaNueva;
           insertarEnPadre(_nodoActual->getPadre(), nroNuevoNodo, claveAlPadre);
                                          delete(claveAlPadre);
           return true;
        default:
           return false;
       }
  }
  else {
   FileDirIndexNode* nuevaRaiz;
   int idH1;
   int idH2;
   FileDirLeafNode* nuevaHoja;
   FileDirLeafNode* nuevaHoja2;
   switch (_raiz->insert(key,reference, &clavesArreglo, &arreglo, &bytesArreglo)) {
     case 1:
       return true;
     case 2:
       nuevaHoja = 0;
```

```
        nuevaHoja2 = 0;

        // creo 2 nodos hoja nuevos para hacer el split y hacer que la nueva raiz sea indice
        _raiz->promoteRoot( (VarLenNode**)(&nuevaHoja), (VarLenNode**)(&nuevaHoja2), _nNodos, _nNodos + 1,
clavesArreglo, &arreglo,    bytesArreglo, &claveARaiz);

        _nNodos += 2;
        // escribo las 2 nuevas hojas
        writeNode(nuevaHoja);
        writeNode(nuevaHoja2);

        idH1 = nuevaHoja->getId();
        idH2 = nuevaHoja2->getId();

        nuevaRaiz = new FileDirIndexNode(_raiz->getId(), _raiz->getNivel() + 1, _raiz->getPadre(), idH1, claveARaiz,
idH2);
        delete _raiz;
        _nodoActual = _raiz = nuevaRaiz;
        return true;

        default:
           return false;
      }
            }
   }
  else {
     _raiz = new FileDirLeafNode();
     _nNodos++;
     _nodoActual = _raiz;
     _raiz->insert(key, reference);
   }
   return true;
}

void FileDirBPlusTree::list()
{
   if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
      writeNode(_nodoActual);
      delete _nodoActual;
      _nodoActual = _raiz;
   }

          while(_nodoActual ->getType() != FileDirNode::LEAF){
                  int proximoALeer = (static_cast<FileDirIndexNode*>(_nodoActual))->getHijoIzquierdo();
                  readNode(proximoALeer, (Node**)&_nodoActual);
          }

          int hnoALeer = 0;
          while (hnoALeer != -1) {

                  (static_cast<FileDirLeafNode*>(_nodoActual))->list();

                  hnoALeer = (static_cast<FileDirLeafNode*>(_nodoActual))->getHnoDerecho();

                  if (hnoALeer >= 0)
                          readNode(hnoALeer, (Node**)&_nodoActual);
          }
          return;
}
```

```
bool FileDirBPlusTree::readHeader()
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      _filestr.seekg(0, ios::beg);
      _filestr.read(_buffer, FileDirNode::VARLEN_NODE_SIZE);
      // leo la cantidad de nodos
      memcpy(&_nNodos, nextByte, sizeof(int));
      return true;
   }
   return false;
}

bool FileDirBPlusTree::writeHeader()
{
   if (_filestr.is_open()) {
      // clear buffer
      memset(_buffer,'\0', FileDirNode::VARLEN_NODE_SIZE);
      char* nextByte = _buffer;
      // volcar en _buffer la cantidad de nodos
      memcpy(nextByte, &_nNodos, sizeof(int));
      // volcar al archivo
      _filestr.seekg(0, ios::beg);
      _filestr.seekp(0, ios::beg);
      _filestr.write(_buffer, FileDirNode::VARLEN_NODE_SIZE);
      return true;
   }
   return false;
}

bool FileDirBPlusTree::readRoot()
{
   if (_filestr.is_open()) {
      _filestr.seekg(FileDirNode::VARLEN_NODE_SIZE, ios::beg);
      _filestr.read(_buffer, FileDirNode::VARLEN_NODE_SIZE);

      if (_nNodos > 1)
         _raiz = new FileDirIndexNode();
      else
         _raiz = new FileDirLeafNode();

      _raiz->read(_buffer);
      return true;
   }
   return false;
}

bool FileDirBPlusTree::writeRoot()
{
   if (_filestr.is_open()) {
      if (_raiz != 0)
         _raiz->write(_buffer);

      if (_filestr.fail())
         _filestr.clear();

      _filestr.seekg(FileDirNode::VARLEN_NODE_SIZE, ios::beg);
      _filestr.seekp(FileDirNode::VARLEN_NODE_SIZE, ios::beg);
      _filestr.write(_buffer, FileDirNode::VARLEN_NODE_SIZE);
      return true;
```

```
    }
    return false;
}

bool FileDirBPlusTree::readNode(int id,Node** node)
{
    if (_filestr.is_open()) {
        if ((*node != _raiz) && (*node != 0)) {
            writeNode(*node);
            delete (*node);
        }
        else
            *node = 0;

        _filestr.seekg(FileDirNode::VARLEN_NODE_SIZE * (id + 1), ios::beg);
        _filestr.read(_buffer, FileDirNode::VARLEN_NODE_SIZE);

        int nivelNodo;
        memcpy(&nivelNodo, _buffer, sizeof(int));

        if (nivelNodo != 0)
            (*node) = new FileDirIndexNode();
        else
            (*node) = new FileDirLeafNode();

        (*node)->read(_buffer);
        return true;
    }
    return false;
}

bool FileDirBPlusTree::writeNode(Node* node)
{
    if (_filestr.is_open()) {
        node->write(_buffer);

        if(_filestr.fail())
            _filestr.clear();

        _filestr.seekp(FileDirNode::VARLEN_NODE_SIZE * (node->getId() + 1), ios::beg);

        if (_filestr.fail())
            _filestr.clear();

        _filestr.write(_buffer, FileDirNode::VARLEN_NODE_SIZE);
        return true;
    }
    return false;
}

void FileDirBPlusTree::insertarEnPadre(int NroNodoPadre, int NroNodoHijo,const char* claveAlPadre)
{
   FileDirIndexNode* nuevoIndice = 0;
   FileDirIndexNode* nuevoIndice2 = 0;
             char* arreglo;
             int bytesArreglo;
             int clavesArreglo;
             char* claveASubir;
             char* claveARaiz;
```

```
  if (NroNodoPadre != 0) { // el padre es distinto de la raiz
    readNode(NroNodoPadre, (Node**)&_nodoActual);

    if (_nodoActual->insert(claveAlPadre, NroNodoHijo, &clavesArreglo, &arreglo, &bytesArreglo) == 2) {
      // hago el split
      nuevoIndice = static_cast<FileDirIndexNode*>((static_cast<FileDirLeafNode*>(_nodoActual))->split
                                                                                    (_nNodos, arreglo, bytesArreglo,
clavesArreglo, &claveASubir));

                                delete(arreglo);
      _nNodos++;
      // escribo los nodos en disco
      writeNode(_nodoActual);
      writeNode(nuevoIndice);

      // actualizo el puntero al padre en los hijos del nuevo nodo indice
      actualizarPadre(nuevoIndice);

      int referenciaNuevoNodo = nuevoIndice->getId();
                                int padreNuevo = nuevoIndice->getPadre();

      delete nuevoIndice;

      insertarEnPadre(padreNuevo, referenciaNuevoNodo, claveASubir);

                                delete(claveASubir);
        return;
      }
      return;
    }

    // si llego hasta aca es porque el que esta "rebalsado" es la raiz
    if (_raiz->insert(claveAlPadre, NroNodoHijo, &clavesArreglo, &arreglo, &bytesArreglo) == 2) {
      _raiz->promoteRoot( (VarLenNode**)(&nuevoIndice), (VarLenNode**)(&nuevoIndice2), _nNodos,_nNodos + 1,
clavesArreglo,&arreglo,bytesArreglo,&claveARaiz);

      delete(arreglo);
      writeNode(nuevoIndice);
      writeNode(nuevoIndice2);

      _nNodos += 2;

      actualizarPadre(nuevoIndice);
      actualizarPadre(nuevoIndice2);

                        FileDirIndexNode* nuevaRaiz = new FileDirIndexNode(_raiz->getId(),_raiz->getNivel() + 1,_raiz-
>getPadre(),

          nuevoIndice->getId(),claveARaiz,nuevoIndice2->getId());

      if (nuevoIndice) delete nuevoIndice;
      if (nuevoIndice2) delete nuevoIndice2;
                        _raiz = nuevaRaiz;

                        delete(claveARaiz);

    }
}

bool FileDirBPlusTree::actualizarPadre(FileDirIndexNode* padre)
```

```
{
   // le seteo el padre al hijo izquierdo
   readNode(padre->getHijoIzquierdo(), (Node**)&_nodoActual);
   _nodoActual->setPadre(padre->getId());

   for(int i = 1; i <= padre->getCantClaves(); ++i) {
      readNode(padre->getRef(i), (Node**)&_nodoActual);
      _nodoActual->setPadre(padre->getId());
   }

   return writeNode(_nodoActual);
}

int FileDirBPlusTree::searchPlace(const char* key)
{
   if ((_nodoActual != _raiz) && (_nodoActual != 0)) {
      writeNode(_nodoActual);
      delete _nodoActual;
      _nodoActual = _raiz;
   }

   return searchPlaceRec(key);
}

int FileDirBPlusTree::searchPlaceRec(const char* key)
{
   if (_nodoActual->getType() == VarLenNode::LEAF)
      return _nodoActual->getId();
   else {
      int indice = _nodoActual->search(key);

      if (readNode(indice, (Node**)&_nodoActual))
         return searchPlaceRec(key);

      return -1;
   }
}
```

**FileVersion.h**

```
#ifndef VERSION_H_INCLUDED
#define VERSION_H_INCLUDED

#include <cstdlib>
#include <string>
#include <iostream>
#include <ctime>

class FileVersion
{
public:

        enum t_versionType { MODIFICACION = 0, BORRADO};

   FileVersion();
   FileVersion(int NroVersion, int Original, tm Fecha, const char* User, long int Offset, char TipoArchivo,t_versionType
VersionType);
   ~FileVersion();

   //empaquetadores y desempaquetadores
   void read (char** buffer);
   void write(char* buffer);

   //getters - no hay setters porque no hay modificacion de la info de las versiones
   int                          getNroVersion()  const { return _nroVersion; }
   int                          getOriginal()    const { return _original;   }
   tm                           getFecha()       const { return _fecha;      }
   char*                getUser()        const { return _user;       }
   long int             getOffset()      const { return _offset;     }
   char                 getTipo()        const           { return _tipo;       }
        t_versionType   getVersionType() const { return _versionType;}

   int tamanioEnDisco();

        FileVersion &operator=(const FileVersion &version);
   int operator==(const FileVersion &version) const;
   int operator<(const FileVersion &version) const;

protected:

   int                          _nroVersion;       // numero de la version
   int                          _original;                 // numero de la version que es el ultimo original de este
archivo
   tm                           _fecha;                    // fecha en que se establecio la version
   char*                _user;                     // el usuario que establecio la version
   long int             _offset;           // offset donde se encuentra la version en el archivo que contiene originales
y diffs
   char                 _tipo;                     // tipo de archivo del que se trata la version "b" = binario, "t" =
texto
        t_versionType   _versionType;      // tipo de version modificacion o borrado
};

#endif
```

**FileVersion.cpp**

```
#include "FileVersion.h"

FileVersion::FileVersion()
{
   _nroVersion          = -1;
   _original            = -1;
   _tipo                = 0;
   _offset                      = -1;
   _user                = 0;
        _versionType    = MODIFICACION;
}

FileVersion::FileVersion(int NroVersion, int Original, tm Fecha, const char* User, long int Offset, char
Tipo,t_versionType VersionType)
{
   _nroVersion = NroVersion;
   _original = Original;
   _fecha = Fecha;
   _offset = Offset;
   _tipo = Tipo;
        _versionType = VersionType;

   int tamanio = strlen(User);

   _user = new char[(tamanio + 1) * sizeof(char)];

   memcpy(_user, User, tamanio * sizeof(char));
   _user[tamanio] = 0;
}

FileVersion::~FileVersion()
{
        if(_user);
                delete _user;
}

void FileVersion::write(char* buffer)
{
   int tamanioUsuario;
   //copio el nro de version
   memcpy(buffer,&_nroVersion,sizeof(int));
   buffer += sizeof(int);

   //copio el nº de version original
   memcpy(buffer,&_original,sizeof(int));
   buffer += sizeof(int);

   //copio la fecha de la version
   memcpy(buffer,&_fecha,sizeof(tm));
   buffer += sizeof(tm);

   //copio el offset dentro del archivo de originales y diffs
   memcpy(buffer,&_offset,sizeof(long int));
   buffer += sizeof(long int);

   //copio el tamaño del campo usuario
   tamanioUsuario = strlen(_user);
   memcpy(buffer,&tamanioUsuario,sizeof(int));
```

```
    buffer += sizeof(int);

    //copio el campo usuario
    memcpy(buffer,_user,tamanioUsuario * sizeof(char));
    buffer += tamanioUsuario * sizeof(char);

    //copio el tipo de archivo
    memcpy(buffer,&_tipo,sizeof(char));
    buffer += sizeof(char);

            //tipo de version
            memcpy(buffer,&_versionType,sizeof(t_versionType));
            buffer += sizeof(t_versionType);

    return;
}

void FileVersion::read(char** buffer)
{
    int tamanioUsuario;
    // leo el nro de version
    memcpy(&_nroVersion,*buffer,sizeof(int));
    *buffer += sizeof(int);

    //leo el nº de version original
    memcpy(&_original,*buffer,sizeof(int));
    *buffer += sizeof(int);

    //leo la fecha de la version
    memcpy(&_fecha, *buffer, sizeof(tm));
    *buffer += sizeof(tm);

    //leo el offset de la version dentro del archivo de originales y diffs
    memcpy(&_offset,*buffer,sizeof(int));
    *buffer+= sizeof(int);

    //leo el usuario, previamente leo la longitud de este campo
    memcpy(&tamanioUsuario,*buffer,sizeof(int));
    *buffer += sizeof(int);

    if(_user != 0){
       delete _user;
       _user = 0;
    }
    _user = new char[(tamanioUsuario + 1)*sizeof(char)];    // creo el campo
    memcpy(_user,*buffer,tamanioUsuario * sizeof(char)); // copio los caracteres
    _user[tamanioUsuario] = 0;  // coloco la marca de fin
    *buffer += tamanioUsuario * sizeof(char);

    //leo el tipo de archivo
    memcpy(&_tipo,*buffer,sizeof(char));
    *buffer += sizeof(char);

            //leo el tipo de version
            memcpy(&_versionType,*buffer,sizeof(t_versionType));
            *buffer += sizeof(t_versionType);

    return;
}
```

```
int FileVersion::tamanioEnDisco(){
   int tamanio = sizeof(int);                                   //nro de version
                  tamanio+= sizeof(int);                                   //nro del original
      tamanio+= sizeof(tm);                                    //fecha
                  tamanio+= sizeof(long int);                              //offset
                  tamanio+= sizeof(int);                                   //longitud del atributo nombre usuario

      tamanio+= strlen(_user) * sizeof(char);//usuario
      tamanio+= sizeof(char);                                  //tipo
                  tamanio+= sizeof(t_versionType);          //tipo de version

   return tamanio;
}

FileVersion& FileVersion::operator=(const FileVersion &version)
{
   _nroVersion          = version._nroVersion;
   _original            = version._original;
   _fecha                     = version._fecha;
   _offset                    = version._offset;
   _tipo                = version._tipo;
         _versionType   = version._versionType;

         int tamanio = strlen(version._user);
   delete _user;
         _user = new char[(tamanio + 1) * sizeof(char)];
         strcpy(_user, version._user);
         return *this;
}

int FileVersion::operator==(const FileVersion &version) const
{
  return ((this->_nroVersion != version._nroVersion) ? 0 : 1);
}

int FileVersion::operator<(const FileVersion &version) const
{
  return ((this->_nroVersion < version._nroVersion) ? 1 : 0);
}
```

**FileBlock.h**

```
#ifndef BLOCK_H_INCLUDED
#define BLOCK_H_INCLUDED

#include "FileVersion.h"

#include <cstdlib>
#include <string>
#include <iostream>

#define TAMANIO_ARREGLO_BLOQUE_ARCHIVOS 1004

class FileBlock
{
public:
   static const int TAMANIO_BLOQUE_ARCHIVOS;

   FileBlock(int Numero = -1,int Anterior = -1 ,int Siguiente = -1);
   ~FileBlock();

   bool insertVersion(FileVersion* version);
   bool searchVersion(int nro,FileVersion** version);
   bool searchVersion(int nro);
   void write(char* buffer);
   void read(char* buffer);
   FileVersion* getLastVersion();

   int getFirstVersionNumber();

   void setSiguiente(int Siguiente){   _siguiente = Siguiente; }
   void setAnterior(int Anterior)  {   _anterior = Anterior;   }

   bool hayLugar(FileVersion* version);

   void moveFirst();
   FileVersion* getNext();
   bool hasNext();

   // getters
   int getSiguiente() const        { return _siguiente;     }
   int getAnterior() const         { return _anterior;      }
   int getNumero() const           { return _numero;        }
   int getCantidadVersiones()const { return _cantVersiones; }

   bool getHistory(std::ostream& os);

private:
   int _siguiente;
   int _anterior;
   int _espacioLibre;
   int _cantVersiones;
   int _used;
   int _actualOffset;
   int _numero;

   char _versiones[TAMANIO_ARREGLO_BLOQUE_ARCHIVOS];
};

#endif
```

**FileBlock.cpp**

```
#include "FileBlock.h"
#include "debug.h"

#include <iostream>

using std::string;
using std::cout;
using std::cerr;
using std::endl;

const int FileBlock::TAMANIO_BLOQUE_ARCHIVOS = 1024;

FileBlock::FileBlock(int Numero,int Anterior,int Siguiente)
{
	_espacioLibre = TAMANIO_ARREGLO_BLOQUE_ARCHIVOS;
	_cantVersiones = 0;
	_numero = Numero;
	_anterior = Anterior;
	_siguiente = Siguiente;
	_used = TAMANIO_ARREGLO_BLOQUE_ARCHIVOS - _espacioLibre;
	_actualOffset = 0;
}

FileBlock::~FileBlock()
{
}

bool FileBlock::insertVersion(FileVersion* version)
{
	char* nextByte = _versiones + _used;
	_espacioLibre -= version->tamanioEnDisco();
	version->write(nextByte);
	_cantVersiones++;
	_used = (TAMANIO_ARREGLO_BLOQUE_ARCHIVOS - _espacioLibre);
	_actualOffset = _used;
   return true;
}

void FileBlock::read(char* buffer)
{
	int offset = 0;

	//numero del bloque
	memcpy(&_numero,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(&_anterior,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(&_siguiente,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(&_espacioLibre,buffer + offset, sizeof(int));
	offset += sizeof(int);
```

```
	_used = TAMANIO_ARREGLO_BLOQUE_ARCHIVOS - _espacioLibre;

	//cantidad de versiones
	memcpy(&_cantVersiones,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//el arreglo con las versiones
	memcpy(_versiones,buffer + offset,TAMANIO_ARREGLO_BLOQUE_ARCHIVOS);
	offset += TAMANIO_ARREGLO_BLOQUE_ARCHIVOS;

	return;
}

void FileBlock::write(char* buffer)
{
	int offset = 0;

	//numero del bloque
	memcpy(buffer + offset, &_numero, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(buffer + offset, &_anterior, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(buffer + offset, &_siguiente, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(buffer + offset, &_espacioLibre, sizeof(int));
	offset += sizeof(int);

	//cantidad de versiones
	memcpy(buffer + offset, &_cantVersiones, sizeof(int));
	offset += sizeof(int);

	//arreglo con las versiones
	memcpy(buffer + offset,_versiones, TAMANIO_ARREGLO_BLOQUE_ARCHIVOS);
	offset += TAMANIO_ARREGLO_BLOQUE_ARCHIVOS;

	return;
}

bool FileBlock::hayLugar(FileVersion* version)
{
	return (_espacioLibre >= version->tamanioEnDisco());
}

bool FileBlock::searchVersion(int nro,FileVersion** version)
{
	char* nextByte = _versiones;

	for (int i = 0;i < _cantVersiones; ++i) {
		*version = new FileVersion();

		(*version) ->read(&nextByte);
		_actualOffset = nextByte - _versiones;

		if ((*version)->getNroVersion() == nro){
```

```
                        return true;
                }

                delete (*version);
        }

        return false;
}

bool FileBlock::searchVersion(int nro)
{
        char* nextByte = _versiones;

        for (int i = 0;i < _cantVersiones; ++i) {

                FileVersion* version = new FileVersion();
                version->read(&nextByte);

                _actualOffset = nextByte - _versiones;

                if(version->getNroVersion() == nro) {
                        delete version;
                        return true;
                }

                delete version;
        }

        return false;
}

FileVersion* FileBlock::getLastVersion()
{
        FileVersion* ret = new FileVersion();
        char* nextByte = _versiones;
        int i = 0;
        while (i < _cantVersiones)
        {
                ret->read(&nextByte);
                _actualOffset = nextByte - _versiones;
                i++;
        }

        return ret;
}

int FileBlock::getFirstVersionNumber()
{
        FileVersion* version = new FileVersion();
        char* nextByte = _versiones;
        version->read(&nextByte);
        _actualOffset = nextByte - _versiones;
        int ret = version->getNroVersion();
        delete version;
        return ret;
}

void FileBlock::moveFirst()
{
        _actualOffset = 0;
```

```
	return;
}

FileVersion* FileBlock::getNext()
{
	FileVersion* ret = new FileVersion();
	char* nextByte = _versiones + _actualOffset;
	ret->read(&nextByte);
	_actualOffset = nextByte - _versiones;
	return ret;
}

bool FileBlock::hasNext()
{
	return (_actualOffset < _used);
}

bool FileBlock::getHistory(std::ostream& os)
{
   FileVersion* fv = new FileVersion();
   char* nextByte = _versiones;

   for (int i = 0; i < _cantVersiones; ++i) {
      fv->read(&nextByte);
      string tipoVersion = ((fv->getVersionType() == FileVersion::MODIFICACION) ? "modificacion" : "borrado");
      os << " version: " << fv->getNroVersion() << ", tipo version: " << tipoVersion << ", usuario: "
        << fv->getUser() << ", fecha: " << asctime(&(fv->getFecha()));
   }
   delete fv;
   return true;
}
```

**FileVersionsFile.h**

```
#ifndef VERSIONFILE_H_INCLUDED
#define VERSIONFILE_H_INCLUDED

#include "FileVersion.h"
#include "FileBlock.h"

#include <fstream>
#include <list>
#include <string>

using std::string;

class FileVersionsFile
{
public:
   enum t_status { ERROR = 0, OK, OVERFLOW };

	FileVersionsFile();
	~FileVersionsFile();

	bool create(const string& a_Filename); // crea el archivo
	bool destroy();

	bool open(const string& a_Filename);
	bool close();
```

```
        // trata de insertar las version en el bloque recibido con referencia:
        // - si lo inserta en el bloque -> devuelve OK
        // - si lo inserta en otro bloque porque el bloque cuyo nro se recibe
        //          como referencia se desborda -> devuelve OVERFLOW
        // - si no lo inserta porque esa version ya estaba en el bloque -> devuelve ERROR
        t_status insertVersion(int nroVersion, const char* User, tm Fecha, long int Offset, char Tipo,
FileVersion::t_versionType VersionType, int bloque, int* nroBloqueNuevo);

        // crea un nuevo bloque para insertar la version, es la 1era version de un archivo nuevo
        // en la variable nroBloqueNuevo se devuelve el nro del bloque que se creo para poder
        // ingresarlo en el indice
        bool insertVersion(int nroVersion, const char* User, tm Fecha, long int Offset, char Tipo,
FileVersion::t_versionType VersionType, int* nroBloqueNuevo);

        bool searchVersion(FileVersion** version, int nroVersion,int bloque);
        bool getVersionFrom(int original, int final, int bloque, std::list<FileVersion>& lstVersions);
        int  getLastOriginalVersionNumber(int bloque);
        int  getLastVersionNumber(int bloque);
        bool getLastVersion(FileVersion** version,int bloque);          //devuelve la ultima version del bloque
   bool getHistory(std::ostream& os, int block);

protected:
        bool readBloque(int nroBloque);
        bool writeBloque();
        bool crearBloque(int Anterior = -1, int Siguiente = -1);
        bool readHeader();
        bool writeHeader();

private:
        std::fstream _filestr;      // file descriptor
        int         _cantBloques;
        FileBlock*   _bloqueActual;
        char*       _buffer;       // buffer de lectura-escritura
   string      _filename;
   bool        _isOpen;
};

#endif
```

**FileVersionFile.cpp**

```
#include "FileVersionsFile.h"
#include "debug.h"

using std::ios;
using std::list;

// constructor
FileVersionsFile::FileVersionsFile() : _cantBloques(0), _bloqueActual(0), _isOpen(false)
{
        _buffer = new char[FileBlock::TAMANIO_BLOQUE_ARCHIVOS];
}

FileVersionsFile::~FileVersionsFile()
{
        delete _bloqueActual;
        delete _buffer;
}

bool FileVersionsFile::readHeader()
```

```
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      _filestr.seekg(0, ios::beg);
      _filestr.read(_buffer, FileBlock::TAMANIO_BLOQUE_ARCHIVOS);
      // leo la cantidad de nodos
      memcpy(&_cantBloques, nextByte, sizeof(int));
      nextByte += sizeof(int);

                        return true;
   }
   return false;
}

bool FileVersionsFile::writeHeader()
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      // volcar en _buffer la cantidad de nodos
      memcpy(nextByte, &_cantBloques, sizeof(int));
      nextByte += sizeof(int);
      // volcar al archivo
      _filestr.seekp(0, ios::beg);
      _filestr.write(_buffer, FileBlock::TAMANIO_BLOQUE_ARCHIVOS);
      return true;
   }
   return false;
}

bool FileVersionsFile::readBloque(int nroBloque)
{
         if (_filestr.is_open()) {
                  if(_bloqueActual != 0){
                           writeBloque(); // escribo el bloque actual;
                           delete _bloqueActual;
                  }

                  _bloqueActual = new FileBlock();

                  _filestr.seekg((nroBloque + 1) * FileBlock::TAMANIO_BLOQUE_ARCHIVOS,ios::beg);
                  _filestr.seekp((nroBloque + 1) * FileBlock::TAMANIO_BLOQUE_ARCHIVOS,ios::beg);

                  _filestr.read(_buffer,FileBlock::TAMANIO_BLOQUE_ARCHIVOS);
                  _bloqueActual->read(_buffer);

                  return true;
         }

         return false;
}

bool FileVersionsFile::writeBloque()
{
         if (_filestr.is_open()) {
                  if (_bloqueActual != 0) {
                           _bloqueActual->write(_buffer);      //escribo el bloque en el buffer

                           _filestr.seekg((_bloqueActual->getNumero() + 1) *
FileBlock::TAMANIO_BLOQUE_ARCHIVOS,ios::beg);
                           _filestr.seekp((_bloqueActual->getNumero() + 1) *
```

```
FileBlock::TAMANIO_BLOQUE_ARCHIVOS,ios::beg);

                                _filestr.write(_buffer,FileBlock::TAMANIO_BLOQUE_ARCHIVOS);
                                return true;
                        }
                }
                return false;  // this does not mean an error
}

bool FileVersionsFile::crearBloque(int Anterior, int Siguiente)
{
        if (_filestr.is_open()) {
                writeBloque(); // escribo el bloque actual;
                _bloqueActual = new FileBlock(_cantBloques,Anterior,Siguiente);
                return true;
        }

        return false;
}

bool FileVersionsFile::create(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("creating FileVersionsFile in '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

        if (!_filestr.is_open()) {
                _filestr.open(a_Filename.c_str(), ios::out | ios::binary);
                _filestr.close();
                _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
                _isOpen = _filestr.is_open();
        }

   _cantBloques = 0;
   _isOpen = _isOpen && writeHeader();
   _filename = a_Filename;
   debug("FileVersionsFile creation " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool FileVersionsFile::destroy()
{
   if (_isOpen)
      return false;

   debug("destroying FileVersionsFile '" + _filename + "'\n");
   int ret = remove(_filename.c_str());
   debug("FileVersionsFile destroy " + string((ret == 0) ? "successfull" : "failed") + "\n");
   return ret == 0;
}

bool FileVersionsFile::open(const string& a_Filename)
{
   if (_isOpen)
      return true;

   debug("opening FileVersionsFile '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
```

```
        _isOpen = _filestr.is_open() && readHeader();
    _filename = a_Filename;
    debug("FileVersionsFile open " + string(_isOpen ? "successfull" : "failed") + "\n");
    return _isOpen;
}

bool FileVersionsFile::close()
{
    if (!_isOpen)
        return true;

    debug("closing FileVersionsFile '" + _filename + "'\n");
        _isOpen = _filestr.is_open() && writeHeader() && writeBloque();
    if (_isOpen) {
        _filestr.close();
        _isOpen = _filestr.is_open();
            }
    debug("FileVersionsFile close " + string(!_isOpen ? "successfull" : "failed") + "\n");
    return !_isOpen;
}

bool FileVersionsFile::insertVersion(int nroVersion, const char* User, tm Fecha, long int Offset, char Tipo,
FileVersion::t_versionType VersionType, int* nroBloqueNuevo)
{
        writeBloque();  // ignore return value

        delete _bloqueActual;
        _bloqueActual = new FileBlock(_cantBloques); // creo el nuevo bloque apuntado por el actual
    if (!_bloqueActual)
        return false;

        *nroBloqueNuevo = _cantBloques;// guardo la referencia al nro del bloque nuevo
        _cantBloques++; // incremento la cantidad de bloques del archivo

        //creo la version nueva
        FileVersion* version = new FileVersion(nroVersion, nroVersion, Fecha, User, Offset, Tipo, VersionType);
        if (!_bloqueActual->insertVersion(version))
        return false;
            delete version;
    return true;
}

FileVersionsFile::t_status FileVersionsFile::insertVersion(int nroVersion, const char* User, tm Fecha, long int Offset,
char Tipo, FileVersion::t_versionType VersionType, int bloque, int* nroBloqueNuevo)
{
        readBloque(bloque); // obtengo el bloque
        FileVersion* ultimaVersion = _bloqueActual->getLastVersion();
        int ultimoOriginal;

        if(VersionType == FileVersion::MODIFICACION)
        {
                if(ultimaVersion->getVersionType() != FileVersion::BORRADO)
                        ultimoOriginal    = ultimaVersion->getOriginal();
                else
                        ultimoOriginal = nroVersion;
        }

        else
                ultimoOriginal = -1;
```

```
		delete ultimaVersion;
		FileVersion* nuevaVersion = new FileVersion(nroVersion, ultimoOriginal, Fecha, User, Offset, Tipo,
VersionType);
		if (_bloqueActual->hayLugar(nuevaVersion)) {
			if (!_bloqueActual->searchVersion(nuevaVersion->getNroVersion())) {
				_bloqueActual->insertVersion(nuevaVersion);
				delete nuevaVersion;
				return FileVersionsFile::OK;
			}

			delete nuevaVersion;
			return FileVersionsFile::ERROR;
		}

		FileBlock* bloqueNuevo = new FileBlock(_cantBloques,_bloqueActual->getNumero());
		_cantBloques++;
		_bloqueActual->setSiguiente(bloqueNuevo->getNumero());
		writeBloque();			// escribo el bloque actual modificado
		delete _bloqueActual;		// elimino el bloque actual
		_bloqueActual = bloqueNuevo;	// seteo el bloque actual como el nuevo
		_bloqueActual->insertVersion(nuevaVersion);
		delete nuevaVersion;
		*nroBloqueNuevo = _bloqueActual->getNumero();
		return FileVersionsFile::OVERFLOW;
}

bool FileVersionsFile::searchVersion(FileVersion** version,int nroVersion,int bloque)
{
   if (bloque < 0)
      return false;

		return (readBloque(bloque) && _bloqueActual->searchVersion(nroVersion, version));
}


bool FileVersionsFile::getVersionFrom(int original, int final, int bloque, list<FileVersion>& lstVersions)
{
		// levanto el bloque a partir del que voy a empezar a buscar la original y armar la lista de versiones.
		// podre moverme hacia bloques anteriores o posteriores.

   lstVersions.clear();
		readBloque(bloque);
		bool found = _bloqueActual->searchVersion(original);

   // buscar el bloque conteniendo la version original
		while (!found) {

			int next = -1;
			int first = _bloqueActual->getFirstVersionNumber();

			if (original < first)
				next = _bloqueActual->getAnterior();
			else
				next = _bloqueActual->getSiguiente();

			if (next < 0)
	      return false;
			else {
				readBloque(next);
```

```
                        found = _bloqueActual->searchVersion(original);
                }
        }

    // en este punto, _bloqueActual contiene la version original
        FileVersion* auxVersion;
        _bloqueActual->searchVersion(original, &auxVersion);

        lstVersions.push_back(*auxVersion);
    //delete auxVersion;

    bool end = (auxVersion->getNroVersion() == final);

    // copio las restantes versiones del 1er bloque
    while (_bloqueActual->hasNext() && !end) {
      auxVersion = _bloqueActual->getNext();
      lstVersions.push_back(*auxVersion);
      if (auxVersion->getNroVersion() <= final) {
                        if(auxVersion->getNroVersion() == final)
                                end = true;
                }
                else
                        end = true;
    }

    int next = _bloqueActual->getSiguiente();
    while ( (!end)&&(next >= 0)) {
      if (!readBloque(next))
        return false;
                _bloqueActual->moveFirst();

      while (_bloqueActual->hasNext() && !end) {
        auxVersion = _bloqueActual->getNext();
        if (auxVersion->getNroVersion() >= final)
          end = true;

        if(auxVersion->getNroVersion() <= final)
                                lstVersions.push_back(*auxVersion);
        //delete auxVersion;
      }
                next = _bloqueActual->getSiguiente();
    }

        return true;
}

int FileVersionsFile::getLastOriginalVersionNumber(int bloque)
{
    if (bloque < 0)
      return -1;

    if (!readBloque(bloque))
      return -1;
    FileVersion* aux = _bloqueActual->getLastVersion();
    int ret = aux->getOriginal();
    delete aux;
    return ret;
}
```

```
int FileVersionsFile::getLastVersionNumber(int bloque)
{
   if (bloque < 0)
      return -1;

   if (!readBloque(bloque))
      return -1;

   FileVersion* aux = _bloqueActual->getLastVersion();
   int ret = aux->getNroVersion();
   delete aux;
   return ret;
}

bool FileVersionsFile::getLastVersion(FileVersion** version, int bloque)
{
   if (bloque < 0)
      return false;

         if (!readBloque(bloque))
      return false;

         _bloqueActual->moveFirst();
         *version = 0;

         while (_bloqueActual->hasNext()) {
                  if (*version)
                           delete *version;
                  *version = _bloqueActual->getNext();
         }
   return true;
}

bool FileVersionsFile::getHistory(std::ostream& os, int block)
{
   readBloque(block);
   do {
      _bloqueActual->getHistory(os);
   } while (_bloqueActual->getSiguiente() > 0);
   return true;
}
```

**File.h**

```
#ifndef FILE_H_INCLUDED
#define FILE_H_INCLUDED

#include <string>
#include <cstdlib>

class File
{
public:
	File();
	File(const char* name, int versionNumber,char type);

	~File();

	char*	getName()			{	return	_name; }
	int	getVersion()	{	return _versionNumber;	}
	char	getType()			{	return _type;	}

	void	setVersion(int Version)	{ _versionNumber = Version; }

	int	getTamanioEnDisco();

	void	read(char** buffer);
	void	write(char* buffer);

private:

	char*	_name;
	int	_versionNumber;
	char	_type;
};

#endif
```

**File.cpp**

```
#include "File.h"

File::File()
{
	_name = 0;
	_versionNumber = -1;
	_type = 0;
}

File::File(const char* name, int versionNumber,char type)
{
	_type = type;
	_versionNumber = versionNumber;

	int tam = strlen(name);
	_name = (char*) malloc((tam+1) * sizeof(char));
	memcpy(_name,name,tam);
	_name[tam] = 0;
}

File::~File()
{
```

```
	if(_name)
		delete _name;
}

int File::getTamanioEnDisco()
{
	int ret = 0;

	ret += sizeof(int);				//indicador de longitud del nombre

	int length = strlen(_name);

	ret += length * sizeof(char);		//el nombre

	ret += sizeof(int);		//version

	ret += sizeof(char);				//tipo

	return ret;
}

void File::write(char* buffer)
{
	int length = strlen(_name);

	memcpy(buffer,&length,sizeof(int));		//indicador del longitud de _name
	buffer += sizeof(int);

	memcpy(buffer,_name,length * sizeof(char));		// _name
	buffer += length * sizeof(char);

	memcpy(buffer,&_versionNumber,sizeof(int));		//_versionNumber
	buffer += sizeof(int);

	memcpy(buffer,&_type,sizeof(char));		//_type
	buffer += sizeof(char);
}

void File::read(char** buffer)
{
	int length;

	memcpy(&length,*buffer,sizeof(int));		//indicador de longitud de _name
	*buffer += sizeof(int);

	if(_name)
		delete _name;

	_name = (char*) malloc((length + 1) * sizeof(char)); //inicializo _name
	memcpy(_name,*buffer,length * sizeof(char));		//_name
	_name[length] = 0;
	*buffer += length * sizeof(char);

	memcpy(&_versionNumber,*buffer,sizeof(int));		//_versionNumber
	*buffer += sizeof(int);

	memcpy(&_type,*buffer,sizeof(char));		//_type
	*buffer += sizeof(char);
}
```

**DirectoryVersion.h**

```
#ifndef DIRECTORY_VERSION_H_INCLUDED
#define DIRECTORY_VERSION_H_INCLUDED

#include "File.h"

#include <list>
#include <string>
#include <ctime>
#include <cstdlib>
#include <iostream>

class DirectoryVersion
{
public:

        enum t_versionType { MODIFICACION = 0, BORRADO};

        DirectoryVersion();
        DirectoryVersion(int NroVersion,const char* User,tm Date,t_versionType Type);

        ~DirectoryVersion();

        void read(char** buffer);
        void write(char* buffer);

        std::list<File>*    getFilesList()      {       return    &_fileLst;                  }

        int                                     getNroVersion()  {       return    _nroVersion;      }
        char*                           getUser()                {       return    _user;                    }
        tm                                      getDate()                {       return    _date;
        }
        size_t                          getCantFile()      {       return    _fileLst.size();}
        t_versionType           getType()                {       return _type;                     }

        void addFile(const char* fileName, int versionNumber,char type);
        void update(const char* fileName, int versionNumber,char type);

        long int tamanioEnDisco();

        bool searchFile(const char* filename, File** file);
        bool searchFile(const char* filename);

private:
        int                                     _nroVersion;      // version number
        char*                           _user;                    // user
        tm                                      _date;                    // date
        std::list<File>         _fileLst;         // list of filenames
        t_versionType           _type;                    // version type
};

#endif
```

**DirectoryVersion.cpp**

```
#include "DirectoryVersion.h"
#include "debug.h"

using std::list;
```

```
using std::string;

DirectoryVersion::DirectoryVersion()
{
	_fileLst.clear();
	_user = 0;
	_nroVersion = -1;
	_type = MODIFICACION;
}

DirectoryVersion::DirectoryVersion(int NroVersion,const char* User,tm Date,t_versionType Type)
{
	_fileLst.clear();
	int tam = strlen(User);
	_nroVersion = NroVersion;
	_user = new char[(tam + 1) * sizeof(char)];
	memcpy(_user,User,tam);
	_user[tam] = 0;
	_date = Date;
	_type = Type;
}

DirectoryVersion::~DirectoryVersion()
{
	_fileLst.clear();

	if (_user)
		delete _user;
}

void DirectoryVersion::addFile(const char* fileName, int versionNumber,char type)
{
	File* newFile = new File(fileName, versionNumber,type);

	_fileLst.push_back(*newFile);
}

void DirectoryVersion::write(char* buffer)
{
	memcpy(buffer,&_nroVersion,sizeof(int));   // version number
	buffer += sizeof(int);

	int length = strlen(_user);
	memcpy(buffer,&length,sizeof(int));                    //length of user
	buffer += sizeof(int);

	memcpy(buffer,_user,length * sizeof(char));//user
	buffer += sizeof(char) * length;

	memcpy(buffer,&_date,sizeof(tm));          //date
	buffer += sizeof(tm);

	memcpy(buffer,&_type,sizeof(t_versionType));       //type
	buffer += sizeof(t_versionType);

	int files = _fileLst.size();
	memcpy(buffer,&files,sizeof(int)); //size of list
	buffer += sizeof(int);

	list<File>::iterator it;
```

```
	int offset = 0;

	for (it = _fileLst.begin();it != _fileLst.end(); ++it) {
		it->write(buffer + offset);  // each file
		offset += it->getTamanioEnDisco();
	}
}

void DirectoryVersion::read(char** buffer)
{
	memcpy(&_nroVersion,*buffer,sizeof(int)); //version number
	*buffer += sizeof(int);

	int length;
	memcpy(&length,*buffer,sizeof(int));			//length of user
	*buffer += sizeof(int);

	if(_user)
		delete _user;

	_user = new char[(length + 1) * sizeof(char)];
	memcpy(_user,*buffer,sizeof(char) * length);		//user
	_user[length] = 0;
	*buffer += sizeof(char) * length;

	memcpy(&_date,*buffer,sizeof(tm));					//date
	*buffer += sizeof(tm);

	memcpy(&_type,*buffer,sizeof(t_versionType));		//type
	*buffer += sizeof(t_versionType);

	_fileLst.clear();

	int files;
	memcpy(&files,*buffer,sizeof(int));					//cantidad de archivos
	*buffer += sizeof(int);

	for (int i = 0; i < files; ++i) {
		File* newFile = new File();						//read each file
		newFile->read(buffer);
		_fileLst.push_back(*newFile);
	}
}

long int DirectoryVersion::tamanioEnDisco()
{
	long int size = 0;

	size += sizeof(int);		//_versionNumber
	size += sizeof(int);		//el indicador de longitud del usuario
	int length = strlen(_user);
	size += length * sizeof(char);		//_user
	size += sizeof(tm);		//_date
	size += sizeof(t_versionType);		//_type
	size += sizeof(int);		//list size
	list<File>::iterator it;
	for (it = _fileLst.begin();it != _fileLst.end(); ++it)
		size += it->getTamanioEnDisco();		//each element of the list
```

```
	return size;
}

void DirectoryVersion::update(const char* fileName, int versionNumber, char type)
{
	list<File>::iterator it;

	bool modified = false;

	for (it = _fileLst.begin(); it != _fileLst.end(); ++it) {
		int cmp = strcmp(it->getName(),fileName);
		if ((cmp == 0) && (it->getType() == type)) {
			it->setVersion(versionNumber);
			modified = true;
		}
	}

	if (!modified)
      addFile(fileName, versionNumber, type);
}

bool DirectoryVersion::searchFile(const char* filename, File** file)
{
	list<File>::iterator it;

	for (it = _fileLst.begin(); it != _fileLst.end(); ++it) {
		int cmp = strcmp(filename, it->getName());
		if(cmp == 0) {
			*file = new File(filename, it->getVersion(),it->getType());
			return true;
		}
	}
	return false;
}

bool DirectoryVersion::searchFile(const char* filename)
{
	list<File>::iterator it;

	for (it = _fileLst.begin(); it != _fileLst.end(); ++it) {
		int cmp = strcmp(filename, it->getName());
		if(cmp == 0)
			return true;
	}
	return false;
}
```

**DirectoryBlock.h**

```
#ifndef DIRECTORY_BLOCK_INCLUDED
#define DIRECTORY_BLOCK_INCLUDED

#include "DirectoryVersion.h"

#include <cstdlib>
#include <string>
#include <iostream>

#define TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS 1004
```

```
class DirectoryBlock
{
public:
	static const int TAMANIO_BLOQUE_DIRECTORIOS;

	DirectoryBlock(int Numero = -1, int Anterior = -1, int Siguiente = -1);

	~DirectoryBlock();

	void insertVersion(DirectoryVersion* version);
	bool searchVersion(int nro,DirectoryVersion** version);
	bool searchVersion(int nro);
	void write(char* buffer);
	void read(char* buffer);
	DirectoryVersion* getLastVersion();

	int getSiguiente()		{		return _siguiente;		}
	int getAnterior()		{		return _anterior;		}
	int getNumero()			{		return _numero;			}
	int getCantidadVersiones()	{		return _cantVersiones;	}

	int getFirstVersionNumber();

	void setSiguiente(int Siguiente){	_siguiente = Siguiente;	}
	void setAnterior(int Anterior)	{	_anterior = Anterior;	}

	bool hayLugar(DirectoryVersion* version);

	void moveFirst();
	DirectoryVersion* getNext();
	bool hasNext();

   bool getHistory(std::ostream& os);

protected:
	int _siguiente;
	int _anterior;
	int _espacioLibre;
	int _cantVersiones;
	int _used;
	int _actualOffset;
	int _numero;

	char _versiones[TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS];
};

#endif
```

**DirectoryBlock.cpp**

```
#include "DirectoryBlock.h"
#include "debug.h"

#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

const int DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS = 1024;
```

```
DirectoryBlock::DirectoryBlock(int Numero,int Anterior,int Siguiente)
{
	_espacioLibre = TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS;
	_cantVersiones = 0;
	_numero = Numero;
	_anterior = Anterior;
	_siguiente = Siguiente;
	_used = TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS - _espacioLibre;
	_actualOffset = 0;
}

DirectoryBlock::~DirectoryBlock()
{
}

void DirectoryBlock::insertVersion(DirectoryVersion* version)
{
	char* nextByte = _versiones + _used;

	_espacioLibre -= version->tamanioEnDisco();
	version->write(nextByte);
	_cantVersiones++;
	_used = (TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS - _espacioLibre);
	_actualOffset = _used;

	return;
}

void DirectoryBlock::read(char* buffer)
{
	int offset = 0;

	//numero del bloque
	memcpy(&_numero,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(&_anterior,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(&_siguiente,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(&_espacioLibre,buffer + offset, sizeof(int));
	offset += sizeof(int);

	_used = TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS - _espacioLibre;

	//cantidad de versiones
	memcpy(&_cantVersiones,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//el arreglo con las versiones
	memcpy(_versiones,buffer + offset,TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS);
	offset += TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS;

	return;
```

```
}

void DirectoryBlock::write(char* buffer)
{
	int offset = 0;

	//numero del bloque
	memcpy(buffer + offset, &_numero, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(buffer + offset, &_anterior, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(buffer + offset, &_siguiente, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(buffer + offset, &_espacioLibre, sizeof(int));
	offset += sizeof(int);

	//cantidad de versiones
	memcpy(buffer + offset, &_cantVersiones, sizeof(int));
	offset += sizeof(int);

	//arreglo con las versiones
	memcpy(buffer + offset,_versiones, TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS);
	offset += TAMANIO_ARREGLO_BLOQUE_DIRECTORIOS;

	return;
}

bool DirectoryBlock::hayLugar(DirectoryVersion* version)
{
	return (_espacioLibre >= version->tamanioEnDisco());
}

bool DirectoryBlock::searchVersion(int nro,DirectoryVersion** version)
{
	char* nextByte = _versiones;

	for (int i = 0;i < _cantVersiones; ++i) {
		*version = new DirectoryVersion();

		(*version) ->read(&nextByte);
		_actualOffset = nextByte - _versiones;

		if ((*version)->getNroVersion() == nro){
			return true;
		}

		delete (*version);
	}

	return false;
}

bool DirectoryBlock::searchVersion(int nro)
{
```

```
	char* nextByte = _versiones;

	for (int i = 0;i < _cantVersiones; ++i) {

		DirectoryVersion* version = new DirectoryVersion();
		version->read(&nextByte);

		_actualOffset = nextByte - _versiones;

		if(version->getNroVersion() == nro) {
			delete version;
			return true;
		}

		delete version;
	}

	return false;
}

DirectoryVersion* DirectoryBlock::getLastVersion()
{
	DirectoryVersion* ret = new DirectoryVersion();
	char* nextByte = _versiones;
	int i = 0;
	while (i < _cantVersiones)
	{
		ret->read(&nextByte);
		_actualOffset = nextByte - _versiones;
		i++;
	}

	return ret;
}

int DirectoryBlock::getFirstVersionNumber()
{
	DirectoryVersion* version = new DirectoryVersion();
	char* nextByte = _versiones;
	version->read(&nextByte);
	_actualOffset = nextByte - _versiones;
	int ret = version->getNroVersion();
	delete version;
	return ret;
}

void DirectoryBlock::moveFirst()
{
	_actualOffset = 0;
	return;
}

DirectoryVersion* DirectoryBlock::getNext()
{
	DirectoryVersion* ret = new DirectoryVersion();
	char* nextByte = _versiones + _actualOffset;
	ret->read(&nextByte);
	_actualOffset = nextByte - _versiones;
	return ret;
}
```

```
bool DirectoryBlock::hasNext()
{
        return (_actualOffset < _used);
}

bool DirectoryBlock::getHistory(std::ostream& os)
{
   DirectoryVersion* dv = new DirectoryVersion();
   char* nextByte = _versiones;

   for (int i = 0; i < _cantVersiones; ++i) {
      dv->read(&nextByte);
      std::string tipoVersion = ((dv->getType() == DirectoryVersion::MODIFICACION) ? "modificacion" : "borrado");
      os << " version: " << dv->getNroVersion() << ", tipo version: " << tipoVersion << ", usuario: "
        << dv->getUser() << ", fecha: " << asctime(&(dv->getDate()));
   }
   delete dv;
   return true;
}
```

**DirectoryVersionsFile.h**

```
#ifndef DIRECTORY_VERSIONS_FILE_INCLUDED
#define DIRECTORY_VERSIONS_FILE_INLCUDED

#include "DirectoryVersion.h"
#include "DirectoryBlock.h"
#include "debug.h"

#include <fstream>
#include <list>
#include <string>

using std::string;

class DirectoryVersionsFile
{
public:
   enum t_status { ERROR = 0, OK, OVERFLOW };

        DirectoryVersionsFile();
        ~DirectoryVersionsFile();

        bool create(const string& a_Filename); // crea el archivo
        bool open(const string& a_Filename);
        bool close();
        bool destroy();

        // trata de insertar las version en el bloque recibido con referencia:
        // - si lo inserta en el bloque -> devuelve OK
        // - si lo inserta en otro bloque porque el bloque cuyo nro se recibe
        //          como referencia se desborda -> devuelve OVERFLOW
        // - si no lo inserta porque esa version ya estaba en el bloque -> devuelve ERROR
        t_status insertVersion(DirectoryVersion* newVersion,int bloque,int* nroBloqueNuevo);

        // crea un nuevo bloque para insertar la version, es la 1? version de un archivo nuevo
        // en la variable nroBloqueNuevo se devuelve el nro del bloque que se creo para poder
        // ingresarlo en el indice
        void insertVersion(DirectoryVersion* newVersion, int* nroBloqueNuevo);
```

```
        bool searchVersion(DirectoryVersion** version, int nroVersion,int bloque);
        bool getVersion(int versionNumber,int bloque,DirectoryVersion** version);
        int getLastVersionNumber(int bloque);
    bool getHistory(std::ostream& os, int block);

protected:
        // file descriptor
        std::fstream _filestr;

        int _cantBloques;
        DirectoryBlock* _bloqueActual;

        // buffer de lectura-escritura
        char*     _buffer;
        string  _filename;
    bool    _isOpen;

        bool readBloque(int nroBloque);
        bool writeBloque();
        bool crearBloque(int Anterior = -1, int Siguiente = -1);
        bool readHeader();
        bool writeHeader();
};

#endif
```

**DirectoryVersionsFile.cpp**

```
#include "DirectoryVersionsFile.h"

using std::list;
using std::ios;

DirectoryVersionsFile::DirectoryVersionsFile()
{
        _buffer = new char[DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS];
        _bloqueActual = 0;
        _cantBloques = 0;
        _isOpen = false;
}

DirectoryVersionsFile::~DirectoryVersionsFile()
{
        delete _bloqueActual;
        delete _buffer;
}

bool DirectoryVersionsFile::readHeader()
{
    if (_filestr.is_open()) {
        char* nextByte = _buffer;
        _filestr.seekg(0, ios::beg);
        _filestr.read(_buffer, DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS);
        // leo la cantidad de bloques
        memcpy(&_cantBloques, nextByte, sizeof(int));
        nextByte += sizeof(int);

                return true;
    }
```

```
    return false;
}

bool DirectoryVersionsFile::writeHeader()
{
    if (_filestr.is_open()) {
        char* nextByte = _buffer;
        // volcar en _buffer la cantidad de nodos
        memcpy(nextByte, &_cantBloques, sizeof(int));
        nextByte += sizeof(int);
        // volcar al archivo
        _filestr.seekp(0, ios::beg);
        _filestr.write(_buffer, DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS);
        return true;
    }
    return false;
}

bool DirectoryVersionsFile::readBloque(int nroBloque)
{
        if (_filestr.is_open()) {
                if(_bloqueActual != 0){
                        writeBloque(); // escribo el bloque actual;
                        delete _bloqueActual;
                }

                _bloqueActual = new DirectoryBlock();

                _filestr.seekg((nroBloque + 1) * DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS,ios::beg);
                _filestr.seekp((nroBloque + 1) * DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS,ios::beg);

                _filestr.read(_buffer,DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS);
                _bloqueActual->read(_buffer);

                return true;
        }

        return false;
}

bool DirectoryVersionsFile::writeBloque()
{
        if (_filestr.is_open()) {

                if(_bloqueActual != 0){
                        _bloqueActual->write(_buffer);      //escribo el bloque en el buffer

                        _filestr.seekg((_bloqueActual->getNumero() + 1) *
DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS,ios::beg);
                        _filestr.seekp((_bloqueActual->getNumero() + 1) *
DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS,ios::beg);

                        _filestr.write(_buffer,DirectoryBlock::TAMANIO_BLOQUE_DIRECTORIOS);

                        return true;
                }
                return false;
        }

        return false;
```

```
}

bool DirectoryVersionsFile::crearBloque(int Anterior, int Siguiente)
{

        if (_filestr.is_open()) {
                writeBloque(); // escribo el bloque actual;
                _bloqueActual = new DirectoryBlock(_cantBloques,Anterior,Siguiente);
                return true;
        }

        return false;
}

bool DirectoryVersionsFile::create(const string& a_Filename)
{
   if(_isOpen)
                return false;
        debug("creating DirectoryVersionsFile in '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

        if (!_filestr.is_open()) {
                _filestr.open(a_Filename.c_str(), ios::out | ios::binary);
                _filestr.close();
                _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
                _isOpen = _filestr.is_open();
      }

   _cantBloques = 0;
   _isOpen = _isOpen && writeHeader();
        _filename = a_Filename;
   debug("DirectoryVersionsFile creation " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool DirectoryVersionsFile::open(const string& a_Filename)
{
   if (_isOpen)
      return true;

   debug("opening DirectoryVersionsFile '" + a_Filename + "'\n");
   _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);

        _isOpen = _filestr.is_open() && readHeader();
   _filename = a_Filename;
   debug("DirectoryVersionsFile open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}


bool DirectoryVersionsFile::close()
{
   if (!_isOpen)
      return true;

   debug("closing DirectoryVersionsFile '" + _filename + "'\n");
        _isOpen = _filestr.is_open() && writeHeader() && writeBloque();
   if (_isOpen) {
      _filestr.close();
      _isOpen = _filestr.is_open();
```

```
        }
    debug("DirectoryVersionsFile close " + string(!_isOpen ? "successfull" : "failed") + "\n");
    return !_isOpen;
}

bool DirectoryVersionsFile::destroy()
{
    if (_isOpen)
        return false;

    debug("destroying DirectoryVersionsFile '" + _filename + "'\n");
    int ret = remove(_filename.c_str());
    debug("DirectoryVersionsFile destroy " + string((ret == 0) ? "successfull" : "failed") + "\n");
    return ret == 0;
}

void DirectoryVersionsFile::insertVersion(DirectoryVersion* newVersion, int* nroBloqueNuevo){

        writeBloque(); // escribo el bloque actual
        delete _bloqueActual;
        _bloqueActual = new DirectoryBlock(_cantBloques); // creo el nuevo bloque apuntado por el actual
        *nroBloqueNuevo = _cantBloques;// guardo la referencia al nro del bloque nuevo
        _cantBloques++; // incremento la cantidad de bloques del archivo

        //inserto la nueva version
        _bloqueActual->insertVersion(newVersion);

        return;
}

DirectoryVersionsFile::t_status DirectoryVersionsFile::insertVersion(DirectoryVersion* newVersion,int bloque,int*
nroBloqueNuevo)
{
        readBloque(bloque); // obtengo el bloque
        if (_bloqueActual->hayLugar(newVersion)) {
                if (!_bloqueActual->searchVersion(newVersion->getNroVersion())) {
                        _bloqueActual->insertVersion(newVersion);
                        return DirectoryVersionsFile::OK;
                }

                return DirectoryVersionsFile::ERROR;
        }

        DirectoryBlock* bloqueNuevo = new DirectoryBlock(_cantBloques,_bloqueActual->getNumero());
        _cantBloques++;
        _bloqueActual->setSiguiente(bloqueNuevo->getNumero());
        writeBloque();                  // escribo el bloque actual modificado
        delete _bloqueActual;               // elimino el bloque actual
        _bloqueActual = bloqueNuevo;        // seteo el bloque actual como el nuevo
        _bloqueActual->insertVersion(newVersion);
        *nroBloqueNuevo = _bloqueActual->getNumero();
        return DirectoryVersionsFile::OVERFLOW;
}

bool DirectoryVersionsFile::searchVersion(DirectoryVersion** version,int nroVersion,int bloque)
{
        readBloque(bloque);
        return _bloqueActual->searchVersion(nroVersion,version);
}
```

```
bool DirectoryVersionsFile::getVersion(int versionNumber,int bloque,DirectoryVersion** version)
{
	// levanto el bloque a partir del que voy a empezar a buscar la original y armar la lista de versiones.
	// podre moverme hacia bloques anteriores o posteriores.

	readBloque(bloque);
	bool found = _bloqueActual->searchVersion(versionNumber);

   // buscar el bloque conteniendo la version original
	while (!found) {

		int next = -1;
		int first = _bloqueActual->getFirstVersionNumber();

		if (versionNumber < first)
			next = _bloqueActual->getAnterior();
		else
			next = _bloqueActual->getSiguiente();

		if (next < 0)
	return false;
		else {
			readBloque(next);
			found = _bloqueActual->searchVersion(versionNumber);
		}
	}

   // en este punto, _bloqueActual contiene la version original
	_bloqueActual->searchVersion(versionNumber, version);

	return true;
}

int DirectoryVersionsFile::getLastVersionNumber(int block)
{
   readBloque(block);
   DirectoryVersion* aux = _bloqueActual->getLastVersion();
   int ret = aux->getNroVersion();
   delete aux;
   return ret;
}

bool DirectoryVersionsFile::getHistory(std::ostream& os, int block)
{
   readBloque(block);
   do {
      _bloqueActual->getHistory(os);
   } while (_bloqueActual->getSiguiente() > 0);
   return true;
}
```

**UserBlock.h**

```
#ifndef USERBLOCK_H_INCLUDED
#define USERBLOCK_H_INCLUDED

#include <cstdlib>
#include <string>
#include <iostream>

#define TAMANIO_ARREGLO_BLOQUE_USUARIO 236

class UserBlock
{
public:
   static const int TAMANIO_BLOQUE_USUARIO;

   UserBlock(int Numero = -1,int Anterior = -1 ,int Siguiente = -1);
   ~UserBlock();

   bool insertRef(int ref);
   void write(char* buffer);
   void read(char* buffer);

   void setSiguiente(int Siguiente){   _siguiente = Siguiente; }
   void setAnterior(int Anterior)  {   _anterior = Anterior;   }

   bool hayLugar();

   void moveFirst();
   int  getNext();
   bool hasNext();
   bool moveTo(int refNumber);

   // getters
   int getSiguiente() const        { return _siguiente;     }
   int getAnterior() const         { return _anterior;      }
   int getNumero() const           { return _numero;        }
   int getCantidadReferencias()const { return _cantReferencias; }

private:
   int _siguiente;
   int _anterior;
   int _espacioLibre;
   int _cantReferencias;
   int _used;
   int _actualOffset;
   int _numero;

   char _referencias[TAMANIO_ARREGLO_BLOQUE_USUARIO];
};

#endif
```

**UserBlock.cpp**

```
#include "UserBlock.h"

const int UserBlock::TAMANIO_BLOQUE_USUARIO = 256;

UserBlock::UserBlock(int Numero,int Anterior,int Siguiente)
```

```
{
	_espacioLibre = TAMANIO_ARREGLO_BLOQUE_USUARIO;
	_cantReferencias = 0;
	_numero = Numero;
	_anterior = Anterior;
	_siguiente = Siguiente;
	_used = TAMANIO_ARREGLO_BLOQUE_USUARIO - _espacioLibre;
	_actualOffset = 0;
}

UserBlock::~UserBlock()
{
}

bool UserBlock::insertRef(int ref)
{
	char* nextByte = _referencias + _used;
	_espacioLibre -= sizeof(int);
	memcpy(nextByte, &ref, sizeof(int));
	_cantReferencias++;
	_used = (TAMANIO_ARREGLO_BLOQUE_USUARIO - _espacioLibre);
	_actualOffset = _used;
   return true;
}

void UserBlock::read(char* buffer)
{
	int offset = 0;

	//numero del bloque
	memcpy(&_numero,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(&_anterior,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(&_siguiente,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(&_espacioLibre,buffer + offset, sizeof(int));
	offset += sizeof(int);

	_used = TAMANIO_ARREGLO_BLOQUE_USUARIO - _espacioLibre;

	//cantidad de versiones
	memcpy(&_cantReferencias,buffer + offset, sizeof(int));
	offset += sizeof(int);

	//el arreglo con las versiones
	memcpy(_referencias,buffer + offset,TAMANIO_ARREGLO_BLOQUE_USUARIO);
	offset += TAMANIO_ARREGLO_BLOQUE_USUARIO;

	return;
}

void UserBlock::write(char* buffer)
{
```

```
	int offset = 0;

	//numero del bloque
	memcpy(buffer + offset, &_numero, sizeof(int));
	offset += sizeof(int);

	//anterior
	memcpy(buffer + offset, &_anterior, sizeof(int));
	offset += sizeof(int);

	//siguiente
	memcpy(buffer + offset, &_siguiente, sizeof(int));
	offset += sizeof(int);

	//espacio libre
	memcpy(buffer + offset, &_espacioLibre, sizeof(int));
	offset += sizeof(int);

	//cantidad de versiones
	memcpy(buffer + offset, &_cantReferencias, sizeof(int));
	offset += sizeof(int);

	//arreglo con las versiones
	memcpy(buffer + offset,_referencias, TAMANIO_ARREGLO_BLOQUE_USUARIO);
	offset += TAMANIO_ARREGLO_BLOQUE_USUARIO;

	return;
}

bool UserBlock::hayLugar()
{
	return (_espacioLibre >= (int)sizeof(int));
}

void UserBlock::moveFirst()
{
	_actualOffset = 0;
	return;
}

int UserBlock::getNext()
{
  int ret;
	char* nextByte = _referencias + _actualOffset;
  memcpy(&ret, nextByte, sizeof(int));
  nextByte += sizeof(int);
  _actualOffset = nextByte - _referencias;
  return ret;
}

bool UserBlock::hasNext()
{
	return (_actualOffset < _used);
}

bool UserBlock::moveTo(int refNumber)
{
  int desp = refNumber * sizeof(int);

  if(desp < _used) {
```

```
      _actualOffset = desp;
      return true;
    }

   return false;
}
```

**UsersRegisterFile.h**

```
#ifndef USERSREGISTERFILE_H_INCLUDED
#define USERSREGISTERFILE_H_INCLUDED

#include "UserBlock.h"

#include <fstream>
#include <list>
#include <string>

using std::string;
using std::list;

class UsersRegisterFile
{
public:
    enum t_status { ERROR = 0, OK, OVERFLOW };

            UsersRegisterFile();
            ~UsersRegisterFile();

            bool create(const string& a_Filename); // crea el archivo
            bool destroy();

            bool open(const string& a_Filename);
            bool close();

            // trata de insertar las version en el bloque recibido con referencia:
            // - si lo inserta en el bloque -> devuelve OK
            // - si lo inserta en otro bloque porque el bloque cuyo nro se recibe
            //             como referencia se desborda -> devuelve OVERFLOW
            // - si no lo inserta porque esa version ya estaba en el bloque -> devuelve ERROR
            t_status insertRef(int reference, int bloque, int* nroBloqueNuevo);

            // crea un nuevo bloque para insertar la version, es la 1era version de un archivo nuevo
            // en la variable nroBloqueNuevo se devuelve el nro del bloque que se creo para poder
            // ingresarlo en el indice
            bool insertRef(int reference, int* nroBloqueNuevo);

   list<int> getReferences(int bloque, int cant);
   list<int> getAllReferences(int bloque);

protected:
            bool readBloque(int nroBloque);
            bool writeBloque();
            bool crearBloque(int Anterior = -1, int Siguiente = -1);
            bool readHeader();
            bool writeHeader();

private:
            std::fstream _filestr;       // file descriptor
            int          _cantBloques;
```

```
        UserBlock*   _bloqueActual;
        char*        _buffer;       // buffer de lectura-escritura
  string       _filename;
  bool         _isOpen;
};

#endif
```

**UsersRegisterFile.cpp**

```
#include "UsersRegisterFile.h"

using std::ios;
using std::list;

// constructor
UsersRegisterFile::UsersRegisterFile() : _cantBloques(0), _bloqueActual(0), _isOpen(false)
{
        _buffer = new char[UserBlock::TAMANIO_BLOQUE_USUARIO];
}

UsersRegisterFile::~UsersRegisterFile()
{
        delete _bloqueActual;
        delete _buffer;
}

bool UsersRegisterFile::readHeader()
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      _filestr.seekg(0, ios::beg);
      _filestr.read(_buffer, UserBlock::TAMANIO_BLOQUE_USUARIO);
      // leo la cantidad de nodos
      memcpy(&_cantBloques, nextByte, sizeof(int));
      nextByte += sizeof(int);

                  return true;
   }
   return false;
}

bool UsersRegisterFile::writeHeader()
{
   if (_filestr.is_open()) {
      char* nextByte = _buffer;
      // volcar en _buffer la cantidad de nodos
      memcpy(nextByte, &_cantBloques, sizeof(int));
      nextByte += sizeof(int);
      // volcar al archivo
      _filestr.seekp(0, ios::beg);
      _filestr.write(_buffer, UserBlock::TAMANIO_BLOQUE_USUARIO);
      return true;
   }
   return false;
}

bool UsersRegisterFile::readBloque(int nroBloque)
{
        if (_filestr.is_open()) {
```

```
                if(_bloqueActual != 0){
                        writeBloque(); // escribo el bloque actual;
                        delete _bloqueActual;
                }

                _bloqueActual = new UserBlock();

                _filestr.seekg((nroBloque + 1) * UserBlock::TAMANIO_BLOQUE_USUARIO,ios::beg);
                _filestr.seekp((nroBloque + 1) * UserBlock::TAMANIO_BLOQUE_USUARIO,ios::beg);

                _filestr.read(_buffer,UserBlock::TAMANIO_BLOQUE_USUARIO);
                _bloqueActual->read(_buffer);

                return true;
        }

        return false;
}

bool UsersRegisterFile::writeBloque()
{
        if (_filestr.is_open()) {
                if (_bloqueActual != 0) {
                        _bloqueActual->write(_buffer);      //escribo el bloque en el buffer

                        _filestr.seekg((_bloqueActual->getNumero() + 1) *
UserBlock::TAMANIO_BLOQUE_USUARIO,ios::beg);
                        _filestr.seekp((_bloqueActual->getNumero() + 1) *
UserBlock::TAMANIO_BLOQUE_USUARIO,ios::beg);

                        _filestr.write(_buffer,UserBlock::TAMANIO_BLOQUE_USUARIO);
                        return true;
                }
        }
        return false;  // this does not mean an error
}

bool UsersRegisterFile::crearBloque(int Anterior, int Siguiente)
{
        if (_filestr.is_open()) {
                writeBloque(); // escribo el bloque actual;
                _bloqueActual = new UserBlock(_cantBloques,Anterior,Siguiente);
                return true;
        }

        return false;
}

bool UsersRegisterFile::create(const string& a_Filename)
{
   if (_isOpen)
      return false;

   _filestr.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

        if (!_filestr.is_open()) {
                _filestr.clear();
                _filestr.open(a_Filename.c_str(), ios::out | ios::binary);
                _filestr.close();
                _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
```

```
                _isOpen = _filestr.is_open();
        }

    _cantBloques = 0;
    _isOpen = _isOpen && writeHeader();
    _filename = a_Filename;
    return _isOpen;
}

bool UsersRegisterFile::destroy()
{
    if (_isOpen)
        return false;

    int ret = remove(_filename.c_str());
    return ret == 0;
}

bool UsersRegisterFile::open(const string& a_Filename)
{
    if (_isOpen)
        return true;

    _filestr.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);

        _isOpen = _filestr.is_open() && readHeader();
    _filename = a_Filename;
    return _isOpen;
}

bool UsersRegisterFile::close()
{
    if (!_isOpen)
        return true;

        _isOpen = _filestr.is_open() && writeHeader() && writeBloque();
    if (_isOpen) {
        _filestr.close();
        _isOpen = _filestr.is_open();
            }
    return !_isOpen;
}

bool UsersRegisterFile::insertRef(int reference, int* nroBloqueNuevo)
{
        writeBloque();  // ignore return value

        delete _bloqueActual;
        _bloqueActual = new UserBlock(_cantBloques); // creo el nuevo bloque apuntado por el actual
    if (!_bloqueActual)
        return false;

        *nroBloqueNuevo = _cantBloques;// guardo la referencia al nro del bloque nuevo
        _cantBloques++; // incremento la cantidad de bloques del archivo

        //creo la version nueva
        if (!_bloqueActual->insertRef(reference))
        return false;
   return true;
}
```

```
UsersRegisterFile::t_status UsersRegisterFile::insertRef(int reference, int bloque, int* nroBloqueNuevo)
{
        readBloque(bloque); // obtengo el bloque

        if (_bloqueActual->hayLugar()) {
                if(_bloqueActual->insertRef(reference))
                  return UsersRegisterFile::OK;

                return UsersRegisterFile::ERROR;
        }

        UserBlock* bloqueNuevo = new UserBlock(_cantBloques,_bloqueActual->getNumero());
        _cantBloques++;
        _bloqueActual->setSiguiente(bloqueNuevo->getNumero());
        writeBloque();              // escribo el bloque actual modificado
        delete _bloqueActual;           // elimino el bloque actual
        _bloqueActual = bloqueNuevo;    // seteo el bloque actual como el nuevo
        _bloqueActual->insertRef(reference);
        *nroBloqueNuevo = _bloqueActual->getNumero();
        return UsersRegisterFile::OVERFLOW;
}

list<int> UsersRegisterFile::getReferences(int bloque, int cant)
{
  list<int> ret;

  readBloque(bloque);

  if(_bloqueActual->getCantidadReferencias() >= cant) {
    _bloqueActual->moveTo(_bloqueActual->getCantidadReferencias() - cant);

    while(_bloqueActual->hasNext()) {
      int ref_a_insertar = _bloqueActual->getNext();
      ret.push_back(ref_a_insertar);
    }

    return ret;
  }

  int restantes = cant - _bloqueActual->getCantidadReferencias();
  int indice = restantes;

  int proximo_a_leer = _bloqueActual->getAnterior();

  while( (restantes > 0) && (proximo_a_leer >= 0) )
  {
    readBloque(proximo_a_leer);
    if(restantes <= _bloqueActual->getCantidadReferencias())
      indice = _bloqueActual->getCantidadReferencias() - restantes;

    restantes -= _bloqueActual->getCantidadReferencias();

    if(restantes > 0)
      proximo_a_leer = _bloqueActual->getAnterior();
  }

  do {
    _bloqueActual->moveTo(indice);
```

```
      while(_bloqueActual->hasNext()) {
        int ref_a_insertar = _bloqueActual->getNext();
        ret.push_back(ref_a_insertar);
      }

      proximo_a_leer = _bloqueActual->getSiguiente();
      if (proximo_a_leer >= 0)
        readBloque(proximo_a_leer);

      indice = 1;
    } while (proximo_a_leer >= 0);

    return ret;
}

list<int> UsersRegisterFile::getAllReferences(int bloque)
{
    list<int> ret;
    readBloque(bloque);

    while(_bloqueActual->getAnterior() >= 0)
        readBloque(_bloqueActual->getAnterior());

    while (true) {
        _bloqueActual->moveFirst();

        while(_bloqueActual->hasNext()) {
            int ref_a_insertar = _bloqueActual->getNext();
            ret.push_back(ref_a_insertar);
        }

        if(_bloqueActual->getSiguiente() < 0)
            break;
        else
            readBloque(_bloqueActual->getSiguiente());
    }

    return ret;
}
```

**Container.h**

```
#ifndef CONTAINER_H_INCLUDED
#define CONTAINER_H_INCLUDED

#include <fstream>
#include <string>

using std::fstream;
using std::string;

class Container
{
public:
   Container() : _isOpen(false) {};

   bool create(const string& a_Filename);
   bool destroy();

   bool open(const string& a_Filename);
   bool close();

   long int append(std::ifstream& is);
   bool get(long int offset, std::ofstream& fs);

private:
   bool    _isOpen;
   fstream _fstream;
   string  _filename;
};

#endif
```

**Container.cpp**

```
#include "Container.h"
#include "debug.h"

using std::ios;

bool Container::create(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("creating Container in '" + a_Filename + "'\n");
   _fstream.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

         if (!_fstream) {
                  _fstream.open(a_Filename.c_str(), ios::out | ios::binary);
                  _fstream.close();
                  _fstream.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
         }

   _isOpen = _fstream.is_open();
   debug("Container creation " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool Container::destroy()
```

```
{
   debug("destroying Container in '" + _filename + "'\n");
   int ret = remove(_filename.c_str());
   debug("Container destroy " + string((ret == 0) ? "successfull" : "failed") + "\n");
   return ret == 0;
}

bool Container::open(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("opening Container in '" + a_Filename + "'\n");
   _fstream.open(a_Filename.c_str(), ios::binary | ios::in | ios::out);
   _isOpen = _fstream.is_open();
   _filename = a_Filename;
   debug("Container open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool Container::close()
{
   if (!_isOpen)
      return true;

   _fstream.close();
   _isOpen = _fstream.is_open();
   return !_isOpen;
}

long int Container::append(std::ifstream& is)
// returns the offset where is was written
{
   if (!_fstream.is_open())
      return -1;

   _fstream.seekg(0, ios::end);
   _fstream.seekp(0, ios::end);

         // get the offset where the archive is copied
   long int pos = _fstream.tellp();

         if (pos < 0) pos = 0;
         is.seekg(0, ios::end);

         long int tamanio = is.tellg();

         is.seekg(0,ios::beg);

   // write length of is
   _fstream.write((char*)&tamanio, sizeof(tamanio));

   // write is into container
   int amount = 0;
   int bufferSize = 100;
   char* buf = new char[bufferSize * sizeof(char)];
   if (!buf) return -1;

   do {
      is.read(buf, bufferSize);
```

```
        amount = is.gcount();
        _fstream.write(buf, amount);
    } while (amount == bufferSize);

    delete buf;
    return pos;
}


bool Container::get(long int offset, std::ofstream& fs)
// offset points to a long int containing the length of the block of text
// read that block and return it
{
    if(!_fstream.is_open()) return false;

    _fstream.seekg(offset, ios::beg);
    _fstream.seekp(offset, ios::beg);

    // write is into container
    int bufferSize = 100;
    char* buf = new char[bufferSize * sizeof(char)];
    if (!buf) return false;

    int fileSize = 0;
    int bytesRead = 0;

    _fstream.read((char*)&fileSize, sizeof(fileSize));

    while (bytesRead < fileSize) {

                        if( (fileSize - bytesRead) >= bufferSize )
                                _fstream.read(buf, bufferSize);
                        else
                                _fstream.read(buf,fileSize - bytesRead);

        int amount = _fstream.gcount();

        fs.write(buf, amount);
        bytesRead += amount;
    }

    delete buf;
    return true;
}
```

**DateLog.h**

```
#ifndef DATELOG_H_INCLUDED
#define DATELOG_H_INCLUDED

#include <fstream>
#include <string>

using std::string;
using std::fstream;

class DateLog
{
public:
   DateLog() : _isOpen(false) {};

   bool create(const string& a_Filename);
   bool destroy();

   bool open(const string& a_Filename);
   bool close();

   long int append(const string& a_Username, const string& a_Date, const string& a_Version, const string&
a_Filename);
   bool showDate(const string& a_Date, int offset);
   bool show(int offset);
   bool showAll();

private:
   bool    _isOpen;
   fstream _fstream;
   string  _filename;
};

#endif
```

**DateLog.cpp**

```
#include "DateLog.h"
#include "debug.h"

#include <iostream>

using std::ios;
using std::cout;
using std::cerr;
using std::endl;

bool DateLog::create(const string& a_Filename)
{
   if (_isOpen)
      return false;

   debug("creating DateLog in '" + a_Filename + "'\n");
   _fstream.open(a_Filename.c_str(), ios::out | ios::in | ios::binary);

         if (!_fstream) {
               _fstream.open(a_Filename.c_str(), ios::out | ios::binary);
               _fstream.close();
               _fstream.open(a_Filename.c_str(), ios::in | ios::out | ios::binary);
```

```
            }

    _isOpen = _fstream.is_open();
    debug("DateLog creation " + string(_isOpen ? "successfull" : "failed") + "\n");
    return _isOpen;
}

bool DateLog::destroy()
{
    debug("destroying DateLog in '" + _filename + "'\n");
    int ret = remove(_filename.c_str());
    debug("DateLog destroy " + string((ret == 0) ? "successfull" : "failed") + "\n");
    return ret == 0;
}

bool DateLog::open(const string& a_Filename)
{
    if (_isOpen)
        return false;

    debug("opening DateLog in '" + a_Filename + "'\n");
    _fstream.open(a_Filename.c_str(), ios::binary | ios::in | ios::out);
    _isOpen = _fstream.is_open();
    _filename = a_Filename;
    debug("DateLog open " + string(_isOpen ? "successfull" : "failed") + "\n");
    return _isOpen;
}

bool DateLog::close()
{
    if (!_isOpen)
        return true;

    _fstream.close();
    _isOpen = _fstream.is_open();
    return !_isOpen;
}

long int DateLog::append(const string& a_Username, const string& a_Date, const string& a_Version, const string&
a_Filename)
// returns the offset where is was written
{
    if (!_fstream.is_open())
        return -1;

    _fstream.seekg(0, ios::end);
    _fstream.seekp(0, ios::end);

            // get the offset where the archive is copied
   long int pos = _fstream.tellp();

            if (pos < 0)
        pos = 0;

   _fstream << a_Date << "\t" << a_Filename << "\t" << a_Version << "\t" << a_Username << "\n";

   return pos;
}

bool DateLog::showDate(const string& a_Date, int offset)
```

```
{
  if(!_fstream.is_open())
    return false;

  _fstream.seekg(offset,ios::beg);
  _fstream.seekp(offset,ios::beg);

  //comienzo a mostrar, al menos va a haber una linea
  string actualDate;
  string strLine;
  char line[1024];
  do{
    _fstream.getline(line,1023);
    strLine = line;
    actualDate = strLine.substr(0,10);

    if(actualDate == a_Date)
      cout << strLine << endl;
  }while((actualDate == a_Date)&&(!_fstream.eof()));

  return true;
}

bool DateLog::show(int offset)
{
   if (!_fstream.is_open())
      return false;

   _fstream.seekg(offset, ios::beg);
   _fstream.seekp(offset, ios::beg);

   string line;
   getline(_fstream, line);

   if (_fstream.fail())
      return false;

   cout << line << endl;
   return true;
}

bool DateLog::showAll()
{
   if (!_fstream.is_open())
      return false;

   _fstream.seekg(0, ios::beg);
   _fstream.seekp(0, ios::beg);

   string line;
   getline(_fstream, line);
   while (!_fstream.eof()) {
      cout << line << endl;
      getline(_fstream, line);
   }

   return true;
}
```

**VersionManager.h**

```
#ifndef VERSION_MANAGER_H_INCLUDED
#define VERSION_MANAGER_H_INCLUDED

#include "FileDirBPlusTree.h"
#include "Container.h"
#include "FileVersion.h"
#include "DirectoryVersion.h"
#include "FileVersionsFile.h"
#include "DirectoryVersionsFile.h"
#include "helpers.h"
#include "FixLenBPlusTree.h"
#include "DateLog.h"
#include "VarLenBPlusTree.h"
#include "UsersRegisterFile.h"

#include <ctime>
#include <string>
#include <list>
#include <sys/stat.h>

using std::string;

class VersionManager
{
public:
   // static members
   static const string FILE_INDEX_FILENAME;
   static const string FILE_VERSION_FILENAME;

   static const string DIR_INDEX_FILENAME;
   static const string DIR_VERSION_FILENAME;

   static const string TXT_DIFFS_FILENAME;
   static const string BIN_DIFFS_FILENAME;

   static const string DIR_CONTAINER_FILENAME;

   static const string DATE_INDEX_FILENAME;
   static const string DATE_LOG_FILENAME;

   static const string USERS_INDEX_FILENAME;
   static const string USERS_REGISTER_FILENAME;

   static const int VERSION_DIGITS;

   // constructor
   VersionManager(const string& a_Almacen, const string& a_Repository);

   bool create();
   bool destroy();

   bool open();
   bool close();

   bool isOpen() const { return _isOpen; }

   bool addFile(int repositoryVersion,const string& repositoryName, const string& a_Filename, const string& a_User,
time_t a_Date, char a_Type);
```

```
    bool addDirectory(int repositoryVersion, const string& repositoryName, const string& a_Directoryname, const
string& a_User, time_t a_Date);
    bool add(int repositoryVersion,const string& repositoryName, const string& a_Target, const string& a_User, time_t
a_Date, t_filetype       a_Type);
    bool addRec(const string& a_Target, int componenteALeer, const string& pathActual,const string& repositoryName,
int repositoryVersion,       int cantComponentesPath, const      string& a_Username, time_t a_Date, t_filetype a_Type);

    bool getFile(const string& a_TargetDir, const string& a_Filename, const string& a_Version,const string&
repositoryName);
    bool getDirectory(const string& a_TargetDir, const string& pathToFile, const string& a_Path, const string&
a_DirName, const string& a_Version,const string& repositoryName);
    bool get(const string& a_Version, const string& a_Target,const string& repositoryName, const string&
a_TargetDestiny);
    bool getDiffByDate(std::ostream& os, const string& a_Date);
    bool getHistory(std::ostream& is, const string& a_Filename);

    bool getDiff(std::ostream& os, const string& a_VersionA, const string& a_VersionB, const string& a_Target, const
string& repositoryName);
    bool getFileDiff(std::ostream& os, const string& a_VersionA, const string& a_VersionB, const string& a_Filename);
    bool getDirectoryDiff(const string& a_DirName, const string& a_VersionA, const string& a_VersionB, int tabs);
    bool getListOfChanges(std::ostream& os, const string& a_Username, int a_Num);

    bool removeFileOrDirectory(int repositoryVersion, const string& repositoryName, const string& pathActual, const
string& a_User, time_t a_Date);
    bool removeFile(int repositoryVersion, const string& repositoryName, const string& a_Filename, const string&
a_User, time_t a_Date);
    bool removeDirectory(int repositoryVersion, const string& repositoryName, const string& a_Directoryname, const
string& a_User, time_t a_Date);

protected:
    bool buildVersion(std::list<FileVersion>& lstVersions, const string& a_Filename);
    bool buildTextVersion(int bloque, FileVersion* versionBuscada, const string& a_Filename);
    bool getFileVersionAndBlock(int* bloque, FileVersion** versionBuscada, const string& a_Filename, const string&
a_Version);
    bool getDirVersion(DirectoryVersion** versionBuscada, const string& a_Dirname, const string& a_Version);
          bool indexAFile(int repositoryVersion, const string& key, const string& a_User, tm* date, int offset, char
a_Type, FileVersion::t_versionType a_VersionType, int bloque);
    void showAddedDirectory(DirectoryVersion* dirVersion, const string& path, int tabs);
    void showDirectory(DirectoryVersion* dirVersion, const string& path, int tabs);

          bool indexADirectory(int repositoryVersion, const string& key, DirectoryVersion* nuevaVersion, int bloque);

   void log(const string& a_Filename, const string& a_Username, const string& a_Version, time_t a_Date);

private:
    // member variables
    bool              _isOpen;
    string            _almacen;
    string            _repository;

    FileDirBPlusTree      _fileIndex;
    FileVersionsFile      _fileVersions;

    FileDirBPlusTree      _dirIndex;
    DirectoryVersionsFile  _dirVersions;

    Container             _textContainer;
    Container             _binaryContainer;

    FixLenBPlusTree       _dateIndex;
```

```
    DateLog              _dateLog;

    UsersRegisterFile    _usersReg;
    VarLenBPlusTree      _usersIndex;
};

#endif
```

**VersionManager.cpp**

```
#include "VersionManager.h"
#include "debug.h"
#include "helpers.h"
#include <fstream>
#include <list>
#include <iterator>
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>

using std::list;
using std::iterator;

using namespace std;

const string VersionManager::FILE_INDEX_FILENAME    = "file_index.ndx";
const string VersionManager::FILE_VERSION_FILENAME  = "file_versions.ndx";

const string VersionManager::DIR_INDEX_FILENAME     = "dir_index.ndx";
const string VersionManager::DIR_VERSION_FILENAME   = "dir_versions.ndx";

const string VersionManager::TXT_DIFFS_FILENAME     = "txt_diffs.dat";
const string VersionManager::BIN_DIFFS_FILENAME     = "bin_diffs.dat";

const string VersionManager::DIR_CONTAINER_FILENAME  = "bin_diffs.dat";

const string VersionManager::DATE_INDEX_FILENAME    = "date_index.ndx";
const string VersionManager::DATE_LOG_FILENAME      =  "date_log.txt";

const string VersionManager::USERS_INDEX_FILENAME   = "users_index.ndx";
const string VersionManager::USERS_REGISTER_FILENAME = "users_register.ndx";

const int VersionManager::VERSION_DIGITS = 5;

VersionManager::VersionManager(const string& a_Almacen, const string& a_Repository)
                        : _isOpen(false), _almacen(a_Almacen), _repository(a_Repository)
{
}

bool VersionManager::destroy()
{
    debug("destroying VersionManager\n");
    bool ret = close();
    ret = ret && _fileIndex      .destroy() &&
                 _fileVersions   .destroy() &&
                 _textContainer  .destroy() &&
                 _binaryContainer.destroy() &&
                 _dirIndex       .destroy() &&
                 _dirVersions    .destroy() &&
                 _dateIndex      .destroy() &&
```

```
            _dateLog        .destroy() &&
            _usersIndex     .destroy() &&
            _usersReg       .destroy();

   debug("VersionManager destroy " + string(ret ? "successfull" : "failed") + "\n");
   return ret;
}

bool VersionManager::open()
{
   if (_isOpen)
      return true;

   debug("opening VersionManager\n");
   string path = _almacen + "//" + _repository + "//";
   _isOpen = (_fileIndex      .open((path + FILE_INDEX_FILENAME)    .c_str()) &&
            _fileVersions   .open((path + FILE_VERSION_FILENAME)  .c_str()) &&
            _textContainer  .open((path + TXT_DIFFS_FILENAME)     .c_str()) &&
            _binaryContainer.open((path + BIN_DIFFS_FILENAME)     .c_str()) &&
            _dirIndex       .open((path + DIR_INDEX_FILENAME)     .c_str()) &&
            _dirVersions    .open((path + DIR_VERSION_FILENAME)   .c_str()) &&
            _dateIndex      .open((path + DATE_INDEX_FILENAME)    .c_str()) &&
            _dateLog        .open((path + DATE_LOG_FILENAME)      .c_str()) &&
            _usersIndex     .open((path + USERS_INDEX_FILENAME)   .c_str()) &&
            _usersReg       .open((path + USERS_REGISTER_FILENAME).c_str())
           );

   debug("VersionManager open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool VersionManager::close()
{
   if (!_isOpen)
      return true;

   debug("closing VersionManager\n");
   bool ret = _fileIndex      .close() &&
            _fileVersions   .close() &&
            _textContainer  .close() &&
            _binaryContainer.close() &&
            _dirIndex       .close() &&
            _dirVersions    .close() &&
            _dateIndex      .close() &&
            _dateLog        .close() &&
            _usersIndex     .close() &&
            _usersReg       .close();
   debug("VersionManager close " + string((ret) ? "successfull" : "failed") + "\n");

   return ret;
}

bool VersionManager::buildVersion(std::list<FileVersion>& lstVersions, const string& a_Filename)
// generates a file named a_Filename with the required version
{
   std::ofstream osfinal(a_Filename.c_str());
   if (!osfinal.is_open())
      return false;

   _textContainer.get(lstVersions.begin()->getOffset(), osfinal);
```

```
    osfinal.close();

    if (lstVersions.size() > 1) {
        std::list<FileVersion>::const_iterator it;
        it = lstVersions.begin();
        ++it;
        for ( ; it != lstVersions.end(); ++it) {
            // save diff into a temporary file
            string tmpFilename = randomFilename(".tmp_");
            std::ofstream osdiff(tmpFilename.c_str());
            _textContainer.get(it->getOffset(), osdiff);
            osdiff.close();

            // apply diff to last version
            string cmd = "ed -s " + systemFilename(a_Filename) + " < " + tmpFilename;
            if (system(cmd.c_str()) != 0)
                return false;
            cmd = "rm -f " + tmpFilename;
            system(cmd.c_str());
        }
    }
    return true;
}


bool VersionManager::addFile(int repositoryVersion, const string& repositoryName, const string& a_Filename, const
string& a_User, time_t a_Date, char a_Type)
{
    if (!_isOpen)
        return false;

    long int offset;
    string key = repositoryName;

    for(int i = 1; i <= countComponents(a_Filename); ++i)
        key = key + "//" + getComponent(a_Filename,i);

    tm* date = localtime(&a_Date);
    // busco en el indice a ver si esta el archivo
    int bloque = _fileIndex.search(key.c_str());

    if (bloque >= 0) { // el archivo esta en el indice
        FileVersion* ultimaVersion;
        _fileVersions.getLastVersion(&ultimaVersion, bloque);
        if (ultimaVersion->getTipo() != a_Type) {
            delete ultimaVersion;
            return false;
        }

        // analizo si la ultima version fue o no de borrado, si es asi, debo copiar todo el archivo
        if (ultimaVersion->getVersionType() == FileVersion::BORRADO) {
            delete ultimaVersion;   //elimino la ultima version

            std::ifstream is(a_Filename.c_str());
            if (!is)
                return false;
            offset = _textContainer.append(is);
            is.close();
            if (offset == -1)
                return false;
```

```
            return indexAFile(repositoryVersion, key, a_User, date, offset, a_Type, FileVersion::MODIFICACION,bloque);
        }

        delete ultimaVersion;

        if (a_Type == 't') {
            // debo insertar el diff si el archivo ya existe o el original sino,
            // al insertar voy a obtener el valor del offset
            int original = _fileVersions.getLastOriginalVersionNumber(bloque);
            int last     = _fileVersions.getLastVersionNumber(bloque);

            list<FileVersion> lstVersions;
            if (!_fileVersions.getVersionFrom(original, last, bloque, lstVersions)) {
                // debo insertar el archivo completo
                std::ifstream is(a_Filename.c_str());
                if (!is)
                    return false;

                offset = _textContainer.append(is);
                is.close();

                if (offset == -1)
                    return false;
            }
            else {
                string tmpVersionFilename = randomFilename(".tmp_");
                buildVersion(lstVersions, tmpVersionFilename);
                // fsVersion contains the file up to version
                // now we need to generate a diff between the file being commited and the last version

                string tmpDiffFilename = randomFilename(".tmp_");
                string cmd = "diff -e " + tmpVersionFilename + " " + systemFilename(a_Filename) + " > " +
tmpDiffFilename;
                if (system(cmd.c_str()) == -1)
                    return false;
                cmd = "echo w >> " + tmpDiffFilename;
                if (system(cmd.c_str()) == -1)
                    return false;

                bool empty = isEmptyFile(tmpDiffFilename);
                if (!empty) {
                    std::ifstream is(tmpDiffFilename.c_str());
                    offset = _textContainer.append(is);
                    is.close();
                }
                // remove temporary files
                remove(tmpVersionFilename.c_str());
                remove(tmpDiffFilename.c_str());

                if (empty)
                    return false;
            }
            return indexAFile(repositoryVersion, key, a_User, date, offset, a_Type, FileVersion::MODIFICACION, bloque);
        }
        else if (a_Type == 'b') {
            std::ifstream is(a_Filename.c_str());
            if (!is)
                return false;
```

```
            // check if versions differ
            FileVersion* ultimaVersion;
            _fileVersions.getLastVersion(&ultimaVersion, bloque);
            if (ultimaVersion->getVersionType() != FileVersion::BORRADO) {
               string tmpVersion = randomFilename(".tmp_");
               ofstream os(tmpVersion.c_str());
               _binaryContainer.get(offset, os);
               os.close();
               if (!areDifferentFiles(tmpVersion, a_Filename))
                  return false;
            }
            offset = _binaryContainer.append(is);
            is.close();
            if (offset == -1)
               return false;

            if (bloque >= 0) { // el archivo esta en el indice
               return indexAFile(repositoryVersion, key, a_User, date, offset, a_Type,
FileVersion::MODIFICACION,bloque);
            }
         }
      }
      else {
         // debo insertar el archivo completo
         std::ifstream is(a_Filename.c_str());
         if (!is.is_open())
            return false;

         if (a_Type == 't')
            offset = _textContainer.append(is);
         else
            offset = _binaryContainer.append(is);

         is.close();
         if (offset == -1)
            return false;

         int nroNuevoBloque;
         if (!_fileVersions.insertVersion(repositoryVersion, a_User.c_str(), *date, offset, a_Type,
FileVersion::MODIFICACION, &nroNuevoBloque))
            return false;
         key = key + zeroPad(repositoryVersion, VERSION_DIGITS);
         return (_fileIndex.insert(key.c_str(), nroNuevoBloque));
      }

      return false; // never gets here
}

bool VersionManager::addDirectory(int repositoryVersion, const string& repositoryName, const string&
a_Directoryname, const string& a_User, time_t a_Date)
{
      if (!_isOpen)
         return false;

      DIR* dir;
      struct dirent* myDirent;
      DirectoryVersion* nuevaVersion;

      string key = repositoryName;
```

```
    for(int i = 1; i <= countComponents(a_Directoryname); ++i)
        key = key + "//" + getComponent(a_Directoryname,i);

    tm* date = localtime(&a_Date);
    // busco en el indice a ver si esta el directorio
    int bloque = _dirIndex.search(key.c_str());

    if (bloque >= 0) {
        // el directorio esta en el indice
        DirectoryVersion* ultimaVersion;
        int lastVersion  = _dirVersions.getLastVersionNumber(bloque); //obtengo el numero de la ultima version

        _dirVersions.getVersion(lastVersion, bloque, &ultimaVersion);   //obtengo la ultima version

        // creo la nueva version
        nuevaVersion = new
DirectoryVersion(repositoryVersion,a_User.c_str(),*date,DirectoryVersion::MODIFICACION);

        // debo cotejar los cambios que hubo en el directorio (eliminacion y agregado de nuevos archivos/directorios)
        // abro el directorio que quiero "versionar"
        if ((dir = opendir(a_Directoryname.c_str())) == NULL) {
            delete nuevaVersion;
            delete ultimaVersion;

            cerr << "El directorio: " << a_Directoryname << " no existe" << endl;
            return false;
        }

        // obtengo una lista con todos los nombres de los archivos/directorios pertenecientes al directorio que quiero
agregar
        list<string> fileIncludedLst;
        while ((myDirent = readdir(dir)) != NULL) {
            string filename = myDirent->d_name;
            if ((filename.compare(".") != 0) && (filename.compare("..") != 0))
                fileIncludedLst.push_back(filename);
        }

        closedir(dir);

        list<File>* filesLst = ultimaVersion->getFilesList(); // lista con los archivos que pertencian a la ultima version
        list<File>::iterator it_oldFiles;     // iterador sobre la lista de los archivos/directorios de la ultima version
        list<string>::iterator it_newFiles;   // iterador sobre la lista de los archivos/directorios de la nueva version

        bool result = true;

        if (ultimaVersion->getType() != DirectoryVersion::BORRADO) {
            // ahora debo cotejar si hay algun borrado y luego generar la version del archivo

            list<File> filesErased; // lista con los nombres de los archivos/directorios que fueron borrados

            for (it_oldFiles = filesLst->begin(); it_oldFiles != filesLst->end(); ++it_oldFiles) {
                string fname = it_oldFiles->getName();
                bool included = false;

                for (it_newFiles = fileIncludedLst.begin();it_newFiles != fileIncludedLst.end(); ++it_newFiles)
                    if (fname.compare(*it_newFiles) == 0)
                        included = true;

                if (!included) {
                    File* file = new File(it_oldFiles->getName(), it_oldFiles->getVersion(), it_oldFiles->getType());
```

```
            filesErased.push_back(*file);
        }
        else
            nuevaVersion->addFile(it_oldFiles->getName(), it_oldFiles->getVersion(), it_oldFiles->getType());
    }

    list<File>::iterator it_erasedFiles;
    for (it_erasedFiles = filesErased.begin(); it_erasedFiles != filesErased.end(); ++it_erasedFiles) {
        string name = it_erasedFiles->getName();
        string fname = a_Directoryname + "/" + name;

        if (it_erasedFiles->getType() != 'd')
            result = result && removeFile(repositoryVersion, repositoryName, fname, a_User, a_Date);
        else
            result = result && removeDirectory(repositoryVersion, repositoryName, fname, a_User, a_Date);

        if(result)
          log(fname, a_User,toString<int>(repositoryVersion),a_Date);
    }
    filesErased.clear();
}

delete ultimaVersion; // elimino la ultima version, ya no la voy a necesitar

list<string>::iterator it_includedFiles;

if (result)
    for (it_includedFiles = fileIncludedLst.begin(); it_includedFiles != fileIncludedLst.end(); ++it_includedFiles) {
        // versiono todos los archivos/directorios que pertenecen al directorio
        string fname = a_Directoryname + "/" + *it_includedFiles;

        t_filetype ftype = getFiletype(fname);
        char type;

        if (nuevaVersion->searchFile(it_includedFiles->c_str())) {
            bool modified = false;
            if (ftype == DIRECTORY) {
                type = 'd';
                modified = addDirectory(repositoryVersion, repositoryName, fname, a_User, a_Date);
            }
            else if (ftype == TEXT) {
                type = 't';
                modified = addFile(repositoryVersion, repositoryName, fname, a_User, a_Date, type);
            }
            else if (ftype == BINARY) {
                type = 'b';
                modified = addFile(repositoryVersion, repositoryName, fname, a_User, a_Date, type);
            }
            if (modified) {
                nuevaVersion->update((*it_includedFiles).c_str(), repositoryVersion, type);
                log(fname, a_User,toString<int>(repositoryVersion),a_Date);
            }
        }
        else {
            if (ftype == DIRECTORY) {
                type = 'd';
                result = result && addDirectory(repositoryVersion, repositoryName, fname, a_User, a_Date);
            }
            else if (ftype == TEXT) {
                type = 't';
```

```
                result = result && addFile(repositoryVersion, repositoryName, fname, a_User, a_Date, type);
              }
              else if (ftype == BINARY) {
                type = 'b';
                result = result && addFile(repositoryVersion, repositoryName, fname, a_User, a_Date, type);
              }
              else result = false;

              if (result) {  // agrego el archivo al directorio
                nuevaVersion->addFile((*it_includedFiles).c_str(), repositoryVersion, type);
                log(fname, a_User,toString<int>(repositoryVersion),a_Date);
               }
            }
          }

        if (result)
          result = indexADirectory(repositoryVersion, key, nuevaVersion, bloque);

        delete nuevaVersion;
        return result;
    }

    else {   // es la 1ra vez que se va a agregar el directorio, por lo tanto, no debo cotejar los cambios
        nuevaVersion = new
DirectoryVersion(repositoryVersion,a_User.c_str(),*date,DirectoryVersion::MODIFICACION);

        if ((dir = opendir(a_Directoryname.c_str())) == NULL) {
          delete nuevaVersion;
          cerr << "El directorio: " << a_Directoryname << " no existe" << endl;
          return false;
        }

        // obtengo una lista con todos los nombres de los archivos/directorios pertenecientes al directorio que quiero
agregar
        list<string> fileIncludedLst;
        while((myDirent = readdir(dir)) != NULL) {
          string filename = myDirent->d_name;
          if ((filename.compare(".") != 0) && (filename.compare("..") != 0))
            fileIncludedLst.push_back(filename);
        }

        closedir(dir);

        list<string>::iterator it_includedFiles;
        bool result = true;

        for (it_includedFiles = fileIncludedLst.begin(); it_includedFiles != fileIncludedLst.end(); ++it_includedFiles) {
          // versiono todos los archivos/directorios que pertenecen al directorio
          string fname = a_Directoryname + "/" + *it_includedFiles;

          t_filetype ftype = getFiletype(fname);
          char type;
          if (ftype == DIRECTORY) {
            type = 'd';
            result = result && addDirectory(repositoryVersion, repositoryName, fname, a_User, a_Date);
          }
          else if ((ftype == TEXT) || (ftype == BINARY)) {
            type = (ftype == TEXT ? 't' : 'b');
            result = result && addFile(repositoryVersion, repositoryName, fname, a_User, a_Date, type);
          }
```

```
            else result = false;

            if (result) {  // agrego el archivo al directorio
               nuevaVersion->addFile((*it_includedFiles).c_str(), repositoryVersion, type);
               log(fname, a_User,toString<int>(repositoryVersion), a_Date);
             }
         }

       if (result) {
          int nroNuevoBloque;
          _dirVersions.insertVersion(nuevaVersion, &nroNuevoBloque);
          key = key + zeroPad(repositoryVersion, VERSION_DIGITS);
          result = (_dirIndex.insert(key.c_str(), nroNuevoBloque));
       }
       delete nuevaVersion;
       return result;
    }
    return false;
}

bool VersionManager::add(int repositoryVersion, const string& repositoryName, const string& a_Target, const string&
a_Username, time_t a_Date, t_filetype a_Type)
{
    int cantComponentesPath = countComponents(a_Target);
    string pathActual = repositoryName;
    return addRec(a_Target,1, pathActual, repositoryName, repositoryVersion, cantComponentesPath, a_Username,
a_Date, a_Type);
}

bool VersionManager::addRec(const string& a_Target, int componenteALeer, const  string& pathActual, const string&
repositoryName, int repositoryVersion, int cantComponentesPath, const string& a_Username, time_t  a_Date, t_filetype
a_Type)
{
    bool ret = false;
    string key;

    string finalPath = repositoryName;

    for (int i = 1; i <= countComponents(a_Target); ++i)
       finalPath = finalPath + "//" + getComponent(a_Target,i);

    if (pathActual.compare(finalPath) == 0) {
       if ((a_Type == BINARY) || (a_Type == TEXT))
          ret = addFile(repositoryVersion,repositoryName, a_Target, a_Username, a_Date, (a_Type == TEXT ? 't' : 'b'));
       else
          ret = addDirectory(repositoryVersion,repositoryName, a_Target, a_Username,a_Date);

       if(ret)
          log(a_Target, a_Username, toString<int>(repositoryVersion), a_Date);
       return ret;
    }
    else {
       // identifico si hay alguna version del directorio actual (por el que voy siguiendo el camino hasta target)
       int bloque = _dirIndex.search(pathActual.c_str());

       //obtengo la fecha
       tm* date = localtime(&a_Date);

       // creo una nueva version de directorio
       DirectoryVersion* nuevaVersion = new DirectoryVersion(repositoryVersion, a_Username.c_str(), *date,
```

```
DirectoryVersion::MODIFICACION);

        // tomo la componente correspondiente
        string componente = getComponent(a_Target,componenteALeer);

        // defino el tipo de archivo que es la componente leida
        char tipoArchivo;
        string caminoRecorrido = pathActual + "//" + componente;
        if (caminoRecorrido.compare(finalPath) == 0) {
           if (a_Type == TEXT)
              tipoArchivo = 't';
           else if (a_Type == BINARY)
              tipoArchivo = 'b';
           else tipoArchivo = 'd';
        }
        else tipoArchivo = 'd';

        if (bloque >= 0) {
           //obtengo el ultimo nro de version del directorio
           int lastVersionNumber = _dirVersions.getLastVersionNumber(bloque);
           DirectoryVersion* oldVersion;

           //trato de obtener la ultima version del directorio
           if (!_dirVersions.getVersion(lastVersionNumber,bloque,&oldVersion))
              ret = false;
           else {
              ret = addRec(a_Target,componenteALeer + 1,pathActual + "//" + componente, repositoryName,
repositoryVersion, cantComponentesPath, a_Username, a_Date, a_Type);

              if (ret) {
                 //aca tengo que cotejar los cambios que se realizan en la version
                 list<File>* lst = oldVersion->getFilesList();

                 list<File>::iterator it;

                 debug("comienzo a copiar la lista");
                 //copio los archivos que tenia en la version anterior a la nueva para luego actualizarlos
                 for (it = lst->begin();it != lst->end(); ++it)
                    nuevaVersion->addFile(it->getName(), it->getVersion(), it->getType());

                 debug("termine de copiar la lista");
                 debug("elimino la version vieja");
                 //libero el puntero a la ultima version
                 delete oldVersion;

                 //actualizo la version de la componente leida
                 nuevaVersion->update(componente.c_str(), repositoryVersion, tipoArchivo);
                 debug("actualizo la version");

                 ret = indexADirectory(repositoryVersion, key, nuevaVersion, bloque);
              }
           }
        }
        else {
           ret = addRec(a_Target, componenteALeer + 1, pathActual + "//" + componente, repositoryName,
repositoryVersion, cantComponentesPath, a_Username,a_Date, a_Type);
           if (ret) {
              int nuevoBloque;
              nuevaVersion->addFile(componente.c_str(), repositoryVersion, tipoArchivo);
              _dirVersions.insertVersion(nuevaVersion,&nuevoBloque);
```

```
            //genero la clave
            key = pathActual + zeroPad(repositoryVersion,VERSION_DIGITS);
            ret = _dirIndex.insert(key.c_str(),nuevoBloque);
          }
        }
        if(ret)
          log(pathActual, a_Username,toString<int>(repositoryVersion), a_Date);
        delete nuevaVersion;
        return ret;
    }

    return ret; // never gets here
}

bool VersionManager::create()
{
    if (_isOpen)
        return true;

    debug("creating VersionManager for Repositorio '" + _repository + "' in Almacen '" + _almacen + "'\n");
    string path = _almacen + "//" + _repository + "//";
    _isOpen = (_fileIndex      .create((path + FILE_INDEX_FILENAME)     .c_str()) &&
               _fileVersions   .create((path + FILE_VERSION_FILENAME)   .c_str()) &&
               _textContainer  .create((path + TXT_DIFFS_FILENAME)      .c_str()) &&
               _binaryContainer.create((path + BIN_DIFFS_FILENAME)      .c_str()) &&
               _dirIndex       .create((path + DIR_INDEX_FILENAME)      .c_str()) &&
               _dirVersions    .create((path + DIR_VERSION_FILENAME)    .c_str()) &&
               _dateIndex      .create((path + DATE_INDEX_FILENAME)     .c_str()) &&
               _dateLog        .create((path + DATE_LOG_FILENAME)       .c_str()) &&
               _usersIndex     .create((path + USERS_INDEX_FILENAME)    .c_str()) &&
               _usersReg       .create((path + USERS_REGISTER_FILENAME) .c_str())
              );

    debug("VersionManager creation " + string(_isOpen ? "successfull" : "failed") + "\n");
    return _isOpen;
}

bool VersionManager::getFile(const string& a_TargetDir, const string& a_Filename, const string& a_Version,const
string& repositoryName)
{
    if (!_isOpen)
        return false;

    FileVersion* versionBuscada;
    int bloque;

     if (!getFileVersionAndBlock(&bloque, &versionBuscada, a_Filename, a_Version))
        return false;

     int RepNameEnd = a_Filename.find_first_not_of(repositoryName + "//");
     string path = a_Filename;
     path.erase(0,RepNameEnd);

     if (versionBuscada->getVersionType() == FileVersion::BORRADO) { // si la version buscada es una version de
borrado entonces no hago nada
        delete versionBuscada;
        return false;
     }
```

```
char ftype = versionBuscada->getTipo();
ifstream fileToCheck;
char option;
if (ftype == 't') {
    int original = versionBuscada->getOriginal();
    // busco en el indice a ver si esta el archivo
    list<FileVersion> versionsList;
    int final = versionBuscada->getNroVersion();
    delete versionBuscada;
    if (_fileVersions.getVersionFrom(original, final, bloque, versionsList)) {
        //chequeo si el archivo ya existe o no. Si ya existe, debo preguntar si lo reescribo o no.
        fileToCheck.open((a_TargetDir + "//" + path).c_str());
        fileToCheck.close();
        if(fileToCheck.fail())  //si el archivo no existia lo escribo
            return (buildVersion(versionsList, a_TargetDir + "//" + path));
        else {
            do {
                // TODO: desde aca interaccion con usuario!?
                cerr << "El archivo: " << a_TargetDir << "/" << path << " ya existe. Desea sobreescribirlo? S/N" << endl;
                cin >> option;
                cout << endl;
                option = toupper(option);
            } while( (option != 'N') && (option != 'S') );

            if (option == 'S')
                return (buildVersion(versionsList, a_TargetDir + "//" + path));
            else
                return true;
        }
    }
    else
        return false;
}
else if (ftype == 'b') {
    int offset = versionBuscada->getOffset();
    delete versionBuscada;
    //chequeo si el archivo ya existe o no. Si ya existe, debo preguntar si lo reescribo o no.
    fileToCheck.open((a_TargetDir + "//" + path).c_str());
    fileToCheck.close();
    if (fileToCheck.fail()) {  //si el archivo no existia lo escribo
        std::ofstream os((a_TargetDir + "//" + path).c_str());
        if (!os.is_open())
            return false;
        if (!_binaryContainer.get(offset, os))
            return false;
        os.close();
        return true;
    }
    else {  // si el archivo ya existia pregunto si hay que sobreescribirlo
        do {
            // TODO: desde aca interaccion con usuario!?
            cerr << "El archivo: " << a_TargetDir << "/" << path << " ya existe. Desea sobreescribirlo? S/N" << endl;
            cin >> option;
            cout << endl;
            option = toupper(option);
        } while((option != 'N') && (option != 'S'));

        if (option == 'S') {
            // sobreescribo el archivo
            std::ofstream os((a_TargetDir + "//" + path).c_str());
```

```
            if (!os.is_open())
               return false;
            if (!_binaryContainer.get(offset, os))
               return false;
            os.close();
            return true;
         }
         else
            return true;
      }
   }
   return false; // never gets here
}

bool VersionManager::getDirectory(const string& a_TargetDir,const string& pathToFile, const string& a_Path, const
string& a_DirName, const string& a_Version,const string& repositoryName)
{
      string fullDirName = a_Path + "//" + a_DirName;
      if (!_isOpen)
            return false;

      DirectoryVersion* versionBuscada;
      if (!getDirVersion(&versionBuscada,fullDirName, a_Version))
         return false;

      if (versionBuscada->getType() == DirectoryVersion::BORRADO) {
         //no puedo recuperar una version de borrado
         delete versionBuscada;
         return false;
      }

      bool ret = true;    //el valor que voy a devolver

      //obtengo la lista de archivos/directorios del directorio que quiero obtener
      list<File>* filesLst = versionBuscada->getFilesList();
      list<File>::iterator it;

      //si el directorio que voy a obtener no esta creado en el destino --> lo creo
      bool creado = false;

      string currentDir = get_current_dir_name();

      if (chdir((a_TargetDir + "//" + pathToFile + "//" + a_DirName).c_str()) != 0) {
         if (mkdir((a_TargetDir + "//" + pathToFile + "//" + a_DirName).c_str(),0755) != 0) {
            cout << "Imposible crear: " << a_TargetDir << "//" << pathToFile << "//" << a_DirName << endl;
            delete versionBuscada;
            return false;
         }
         else {
            creado = true;
            chdir(currentDir.c_str());
         }
      }

      for(it = filesLst->begin(); it != filesLst->end(); ++it) {
         string FName = it->getName();
         string version_number = toString<int>(it->getVersion());

         if (it->getType() != 'd')
            ret = ret && getFile(a_TargetDir, fullDirName + "//" + FName, version_number, repositoryName);
```

```
            else
                ret = ret && getDirectory(a_TargetDir,pathToFile + "//" + a_DirName, fullDirName, FName, version_number,
repositoryName);
        }

        // si fallo la recuperacion y cree un directorio --> lo elimino
        if ((ret == false) && (creado == true))
            remove((a_TargetDir + "//" + pathToFile + "//" + a_DirName).c_str());

        delete versionBuscada;
        chdir(currentDir.c_str());
        return ret;
}

bool VersionManager::get(const string& a_Version, const string& a_Target,const string& repositoryName, const
string& a_TargetDestiny)
{
    if (!_isOpen)
        return false;

    // voy a tener que la version del directorio que contiene a ese archivo/directorio para cotejar que existe la version que
estoy buscando
    DirectoryVersion* versionDirectorioContenedor;

    // tengo que armar el path del directorio contenerdor al archivo/directorio que estoy buscando
    // para todos los casos el directorio raiz es el repositorio por lo tanto todos los paths de los archivos empiezan con:
    // "nombre_repositorio//" y van seguidos del path correspondiente...

    string searchingPath = repositoryName;

    // voy agregando componente a componente para saber donde debo buscar
    for(int i = 1; i < countComponents(a_Target); ++i)
        searchingPath = searchingPath + "//" + getComponent(a_Target, i);

    // obtengo la version del directorio que contiene al archivo/directorio objetivo con el mismo nro de version que deseo
que tenga el
    // archivo/directorio si es que voy a querer una version en particular o la ultima version del directorio que contiene al
archivo/
    // directorio si esa que no especifique ninguna
    if(!getDirVersion(&versionDirectorioContenedor, searchingPath,a_Version))
        return false;

    if (a_Target != "") {
        if (versionDirectorioContenedor->getType() == DirectoryVersion::BORRADO) {
            delete versionDirectorioContenedor;
            return false;
        }

        string filename = getComponent(a_Target,countComponents(a_Target));

        File* file;
        if (!versionDirectorioContenedor->searchFile(filename.c_str(),&file)) {
            delete versionDirectorioContenedor;
            return false;
        }

        // una vez obtenida la version del directorio voy a tener que armar la estructura de directorios que contienen al
archivo/directorio
        // objetivo dentro del directorio destino para luego "bajar" la version solicitada del archivo/directorio objetivo
```

```
    // aca voy a guardar el path actual para luego volver al directorio de trabajo
    string currentDirectory = get_current_dir_name();

    // empiezo a armar la estructura
    int existentes  = 0;

    // trato de cambiar de directorio al directorio destino para chequear que existe
    if(chdir(a_TargetDestiny.c_str()) != 0) {
       // si el directorio destion no existe -> vuelvo al directorio de trabajo actual y elimino las referencias que tengo en
memoria
       chdir(currentDirectory.c_str());
       delete versionDirectorioContenedor;
       cout << "El directorio elegido como destino no existe." << endl;
       return false;
    }

    chdir(currentDirectory.c_str()); //vuelvo al directorio de trabajo

    //ahora tengo que armar la estructura de directorios a donde va a ir a parar el archivo/directorio objetivo
    int RepNameEnd = searchingPath.find_first_not_of(repositoryName + "//");

    string path = searchingPath;
    path.erase(0,RepNameEnd);
    string pathAuxiliar = a_TargetDestiny;
    int components = countComponents(path);
    if(path.length() > 0) {
       //empiezo a armar la estructura de directorios
       int components = countComponents(path);

       debug("creando directorios \n");

       for(int j = 1; j <= components; ++j) {
          pathAuxiliar = pathAuxiliar + "/" + getComponent(path,j);

          //si no existe el directorio lo creo
          if(chdir(pathAuxiliar.c_str()) != 0) {
             if(mkdir(pathAuxiliar.c_str(),0755) != 0)
                cout << "error al crear: " << pathAuxiliar << endl;
          }
          else {
             existentes++;
             chdir(currentDirectory.c_str());
          }
       }
    }

    bool ret;

    if(file->getType() != 'd')
       ret = getFile(a_TargetDestiny,searchingPath + "//" + filename ,a_Version,repositoryName);
    else
       ret = getDirectory(a_TargetDestiny,path , searchingPath, filename, a_Version,repositoryName);

    if(ret == false) {
       //elimino los directorios que cree para albergar el archivo/directorio que queria obtener
       for(int j = components; j > existentes; --j) {
          remove(pathAuxiliar.c_str());
          int index = pathAuxiliar.find_last_of("/");
          pathAuxiliar.erase(index);
       }
```

```
        }

        delete versionDirectorioContenedor;
        return ret;
    }
    else {
        bool ret = true;
        list<File>* filesLst = versionDirectorioContenedor->getFilesList();
        list<File>::iterator it_files;
        string fname;
        for(it_files = filesLst->begin(); it_files != filesLst->end(); ++it_files) {
            fname = it_files->getName();
            if(it_files->getType() == 'd')
                ret = ret && getDirectory(a_TargetDestiny, "" , searchingPath, fname, a_Version, repositoryName);
            else
                ret == ret && getFile(a_TargetDestiny,searchingPath + "//" + fname, a_Version, repositoryName);
        }

        if (!ret) {
            for(it_files = filesLst->begin(); it_files != filesLst->end(); ++it_files) {
                fname = it_files->getName();
                remove((a_TargetDestiny + "/" + fname).c_str());
            }
        }
        delete versionDirectorioContenedor;
        return ret;
    }
}

bool VersionManager::removeFileOrDirectory(int repositoryVersion, const string& repositoryName, const string&
pathActual, const string& a_User, time_t a_Date)
{
    // TODO
    return false;
}

bool VersionManager::removeFile(int repositoryVersion, const string& repositoryName, const string& a_Filename,
const string& a_User, time_t a_Date)
{
    if (!_isOpen)
        return false;

    string key = repositoryName;
    for(int i = 1; i <= countComponents(a_Filename); ++i)
        key = key + "//" + getComponent(a_Filename,i);

    debug("clave a borrar: "+key+"\n");

    tm* date = localtime(&a_Date);
    // busco en el indice a ver si esta el archivo
    int bloque = _fileIndex.search(key.c_str());

    if (bloque >= 0) { // el archivo esta en el indice, entonces, se puede borrar
        FileVersion* ultimaVersion;
        _fileVersions.getLastVersion(&ultimaVersion, bloque);
        if (ultimaVersion->getVersionType() == FileVersion::BORRADO) {
            // si ya esta borrado no puedo volver a borrarlo
            delete ultimaVersion;
            return false;
        }
```

```
        debug("obtengo el fileType \n");
        char tipoArchivo = ultimaVersion->getTipo();

        delete ultimaVersion;
        log(a_Filename, a_User,toString<int>(repositoryVersion), a_Date);
        return indexAFile(repositoryVersion, key, a_User, date, -1, tipoArchivo, FileVersion::BORRADO, bloque);
    }
    debug("bloque < 0 \n");
    return false;
}

bool VersionManager::removeDirectory(int repositoryVersion, const string& repositoryName, const string&
a_Directoryname, const string& a_User, time_t a_Date)
{
    debug("ingreso en removeDirectory \n");
    if (!_isOpen)
        return false;

    DirectoryVersion* nuevaVersion;
    string key = repositoryName;

    for(int i = 1; i <= countComponents(a_Directoryname); ++i)
        key = key + "//" + getComponent(a_Directoryname,i);

    tm* date = localtime(&a_Date);
    // busco en el indice a ver si esta el directorio
    int bloque = _dirIndex.search(key.c_str());

    if (bloque >= 0) { // el directorio esta en el indice
        DirectoryVersion* ultimaVersion;
        int lastVersion  = _dirVersions.getLastVersionNumber(bloque); //obtengo el numero de la ultima version

        _dirVersions.getVersion(lastVersion, bloque, &ultimaVersion);   //obtengo la ultima version

        debug("bloque >= 0 \n");
        if (ultimaVersion->getType() == DirectoryVersion::BORRADO) {
            //no puedo volver a borrar algo ya borrado
            delete ultimaVersion;
            debug("ultima version fue borrado \n");
            return false;
        }

        bool result = true;
        //creo la nueva version
        nuevaVersion = new DirectoryVersion(repositoryVersion,a_User.c_str(),*date,DirectoryVersion::BORRADO);

        list<File>* filesLst = ultimaVersion->getFilesList();

        list<File>::iterator it_filesToRemove;
        // a cada archivo/directorio de la version anterior les debo hacer un borrado
        for(it_filesToRemove = filesLst->begin();it_filesToRemove != filesLst->end(); ++it_filesToRemove) {

            string fname = it_filesToRemove->getName();
            string fullPath = a_Directoryname + "/" + fname;

            debug("archivo a eliminar: "+fullPath+"\n");
            if(it_filesToRemove->getType() == 'd') {
                result = result && removeDirectory(repositoryVersion,repositoryName,fullPath,a_User,a_Date);
                debug("entro en removeDirectory \n");
```

```
            }
            else {
               result = result && removeFile(repositoryVersion,repositoryName,fullPath,a_User,a_Date);
               debug("entro en removeFile \n");
            }
            if(!result)
               debug("el borrado falla en: "+fullPath+"\n");
         }

         if (!result) {
            debug("fallo el borrado \n");
            delete ultimaVersion;
            delete nuevaVersion;

            return false;
         }

         result = indexADirectory(repositoryVersion, key, nuevaVersion, bloque);

         if(result)
            log(a_Directoryname, a_User,toString<int>(repositoryVersion), a_Date);
         delete nuevaVersion;
         return result;
      }

      // si llega aca es porque no habia una version previa del directorio, por lo tanto, no se puede realizar el borrado
      return false;
   }

   bool VersionManager::getFileVersionAndBlock(int* bloque, FileVersion** versionBuscada, const string& a_Filename,
   const string& a_Version)
   {
      if (a_Version != "") {
         int version = fromString<int>(a_Version);
         *bloque = _fileIndex.searchFileAndVersion(a_Filename.c_str(), version);
         if (!_fileVersions.searchVersion(versionBuscada, version, *bloque)) {
            *bloque = _fileIndex.search(a_Filename.c_str());
            if (!_fileVersions.getLastVersion(versionBuscada, *bloque))
               return false;
            else if ((*versionBuscada)->getNroVersion() > version) {
               return false;
            }
         }
      }
      else {
         *bloque = _fileIndex.search(a_Filename.c_str());
         if (!_fileVersions.getLastVersion(versionBuscada, *bloque))
           return false;
      }
      return true;
   }

   bool VersionManager::getDirVersion(DirectoryVersion** versionBuscada, const string& a_Dirname, const string&
   a_Version)
   {
      int bloque;
      if (a_Version != "") {
         int version = fromString<int>(a_Version);
         bloque = _dirIndex.searchFileAndVersion(a_Dirname.c_str(), version);
         if (bloque < 0)
```

```
            return false;
        if (!_dirVersions.searchVersion(versionBuscada, version, bloque)) {
            bloque = _dirIndex.search(a_Dirname.c_str());
            if (bloque < 0)
                return false;
            version = _dirVersions.getLastVersionNumber(bloque);
            if (!_dirVersions.getVersion(version, bloque, versionBuscada))
                return false;
            else if((*versionBuscada)->getNroVersion() > version) {
                delete (*versionBuscada);
                return false;
            }
        }
    }
    else {
        bloque = _dirIndex.search(a_Dirname.c_str());
        if(bloque < 0) return false;
        int lastVersion = _dirVersions.getLastVersionNumber(bloque);
        if (!_dirVersions.getVersion(lastVersion, bloque, versionBuscada))
            return false;
    }
    return true;
}

bool VersionManager::indexAFile(int repositoryVersion, const string& key, const string& a_User, tm* date, int offset,
char a_Type, FileVersion::t_versionType a_VersionType, int bloque)
{
    int nroNuevoBloque;
    string newKey;

    FileVersionsFile::t_status status = _fileVersions.insertVersion(repositoryVersion, a_User.c_str(), *date, offset,
a_Type, a_VersionType, bloque, &nroNuevoBloque);
    switch (status) {
        case FileVersionsFile::OK :
            return true;
            break;
        case FileVersionsFile::OVERFLOW :
            // tengo que generar la clave a partir de a_File y repositoryVersion
            newKey = key + zeroPad(repositoryVersion, VERSION_DIGITS);
            return _fileIndex.insert(newKey.c_str(), nroNuevoBloque);
            break;
        default:
            return false;
            break;
    }
    return false;
}

bool VersionManager::indexADirectory(int repositoryVersion, const string& key, DirectoryVersion* nuevaVersion,int
bloque)
{
    int nroNuevoBloque;
    string newKey;

    DirectoryVersionsFile::t_status status = _dirVersions.insertVersion(nuevaVersion, bloque, &nroNuevoBloque);
    switch (status) {
      case DirectoryVersionsFile::OK :
        return true;
        break;
      case DirectoryVersionsFile::OVERFLOW :
```

```
        // tengo que generar la clave a partir de a_Directoryname y repositoryVersion
          newKey = key + zeroPad(repositoryVersion, VERSION_DIGITS);
        return _dirIndex.insert(key.c_str(), nroNuevoBloque);
          break;

      default:
        return false;
        break;
  }
  return false;
}

bool VersionManager::buildTextVersion(int bloque, FileVersion* versionBuscada, const string& a_Filename)
{
  int original = versionBuscada->getOriginal();
  int final    = versionBuscada->getNroVersion();

  list<FileVersion> versionsList;
  if (_fileVersions.getVersionFrom(original, final, bloque, versionsList))
    return (buildVersion(versionsList, a_Filename));

  return false;
}

bool VersionManager::getFileDiff(std::ostream& os, const string& a_VersionA, const string& a_VersionB, const
string& a_Filename)
{
  if (!_isOpen)
    return false;

  FileVersion* versionBuscadaA;
  FileVersion* versionBuscadaB;
  int bloqueA, bloqueB;

  if (!getFileVersionAndBlock(&bloqueA, &versionBuscadaA, a_Filename, a_VersionA))
    return false;

  if (!getFileVersionAndBlock(&bloqueB, &versionBuscadaB, a_Filename, a_VersionB)) {
    delete versionBuscadaA;
    return false;
  }

  char ftypeA = versionBuscadaA->getTipo();
  char ftypeB = versionBuscadaB->getTipo();
  if (ftypeA != ftypeB)
    return false;

  string tmpA    = randomFilename(".tmp_");
  string tmpB    = randomFilename(".tmp_");
  string tmpDiff = randomFilename(".tmp_");
  if (ftypeA == 't') {
    bool ret = buildTextVersion(bloqueA, versionBuscadaA, tmpA);
    ret = ret && buildTextVersion(bloqueB, versionBuscadaB, tmpB);
    delete versionBuscadaA;
    delete versionBuscadaB;

    if (ret) {
      // perform diff between tmpA and tmpB
      string cmd = "diff " + tmpA + " " + tmpB + " > " + tmpDiff;
      if (system(cmd.c_str()) == -1)
```

```
            return false;

        ifstream is(tmpDiff.c_str());
        string line;
        getline(is, line);
        while (!is.eof()) {
            os << line << endl;
            getline(is, line);
        }
        is.close();

        // remove temporary files
        remove(tmpA.c_str());
        remove(tmpB.c_str());
        remove(tmpDiff.c_str());
    }
    return ret;
}
else if (ftypeA == 'b') {
    // obtener ambos archivos y compararlos
    int offsetA = versionBuscadaA->getOffset();
    int offsetB = versionBuscadaB->getOffset();
    delete versionBuscadaA;
    delete versionBuscadaB;

    std::ofstream osA(tmpA.c_str());
    std::ofstream osB(tmpB.c_str());
    if (!osA.is_open() || !osB.is_open())
        return false;
    if (!_binaryContainer.get(offsetA, osA) ||
        !_binaryContainer.get(offsetB, osB))
        return false;
    osA.close();
    osB.close();

    string cmd = "cmp " + tmpA + " " + tmpB + "| wc -c > " + tmpDiff;
    if (system(cmd.c_str()) == -1)
        return false;

    ifstream is(tmpDiff.c_str());
    int different;
    is >> different;
    is.close();
    std::ofstream os(tmpDiff.c_str());
    if (different == 0)
        os << "Ambas versiones son identicas.\n";
    else
        os << "Las versiones difieren.\n";
    os.close();

    is.open(tmpDiff.c_str());
    string line;
    getline(is, line);
    while (!is.eof()) {
        os << line << endl;
        getline(is, line);
    }
    is.close();

    // remove temporary files
```

```
        remove(tmpA.c_str());
        remove(tmpB.c_str());
        remove(tmpDiff.c_str());

        return true;
    }
    return false;
}


bool VersionManager::getDirectoryDiff(const string& a_DirName, const string& a_VersionA, const string&
a_VersionB, int tabs)
{
    string espacio;
    for(int i = 1; i <= tabs; ++i)
        espacio = espacio + "\t";

    string dirname = a_DirName;
    int index = dirname.find_last_of("//",dirname.length());
    dirname.erase(0,index + 1);

    bool ret = true;

    DirectoryVersion* versionDirectorioA;
    DirectoryVersion* versionDirectorioB;

    if(!getDirVersion(&versionDirectorioA, a_DirName, a_VersionA))
        return false;

    if(!getDirVersion(&versionDirectorioB, a_DirName, a_VersionB)) {
        delete versionDirectorioA;
        return false;
    }

    list<File>* lstA = versionDirectorioA->getFilesList();
    list<File>* lstB = versionDirectorioB->getFilesList();

    list<File>::iterator it_listA;
    list<File>::iterator it_listB;

    string fname;

    cout << espacio << dirname << endl;

    for(it_listA = lstA->begin(); it_listA != lstA->end(); ++it_listA) {
        bool found = false;
        for (it_listB = lstB->begin(); it_listB != lstB->end(); ++it_listB) {
            if (strcmp(it_listA->getName(),it_listB->getName()) == 0) {
                found = true;
                fname = it_listA->getName();

                if (it_listA->getVersion() != it_listB->getVersion()) {

                    if((it_listA->getType() != 'd') && (it_listB->getType() != 'd')) {
                        FileVersion* file_versionA;
                        FileVersion* file_versionB;

                        int bloqueA;
                        int bloqueB;
```

```
                if(!getFileVersionAndBlock(&bloqueA, &file_versionA, a_DirName + "//" + fname, a_VersionA))
                    break;

                if(!getFileVersionAndBlock(&bloqueB, &file_versionB, a_DirName + "//" + fname, a_VersionB)) {
                    delete file_versionA;
                    break;
                }

                cout << espacio + "    " << fname << " version: " << file_versionA->getNroVersion() << ", tipo: " <<
file_versionA->getTipo()
                     << ", usuario: " << file_versionA->getUser() << "    ";
                cout << espacio + "    " << fname << " version: " << file_versionB->getNroVersion() << ", tipo: " <<
file_versionB->getTipo()
                     << ", usuario: " << file_versionB->getUser() << endl;

                delete file_versionA;
                delete file_versionB;
            }

            else if ((it_listA->getType() == 'd') && (it_listB->getType() == 'd'))
                ret = ret && getDirectoryDiff(a_DirName + "//" + fname, a_VersionA, a_VersionB, tabs + 1);

            else if ((it_listA->getType() == 'd') && (it_listB->getType() != 'd')) {
                DirectoryVersion* dir_version;
                FileVersion* file_version;
                int bloque;

                if (!getDirVersion(&dir_version, a_DirName + "//" + fname, a_VersionA))
                    break;

                if (!getFileVersionAndBlock(&bloque, &file_version, a_DirName + "//" + fname, a_VersionB)) {
                    delete dir_version;
                    break;
                }

                cout << espacio + "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                     << ", usuario: " << dir_version->getUser() << "    ";
                cout << espacio + "    " << fname << " version: " << file_version->getNroVersion() << ", tipo: " <<
file_version->getTipo()
                     << ", usuario: " << file_version->getUser() << endl;

                delete file_version;
                delete dir_version;
            }
            else {
                DirectoryVersion* dir_version;
                FileVersion* file_version;
                int bloque;

                if (!getDirVersion(&dir_version, a_DirName + "//" + fname, a_VersionB))
                    break;

                if (!getFileVersionAndBlock(&bloque, &file_version, a_DirName + "//" + fname, a_VersionA)) {
                    delete file_version;
                    break;
                }

                cout << espacio + "    " << fname << " version: " << file_version->getNroVersion() << ", tipo: " <<
file_version->getTipo()
                     << ", usuario: " << file_version->getUser() << "    ";
```

```
                cout << espacio + "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                    << ", usuario: " << dir_version->getUser() << endl;

                delete file_version;
                delete dir_version;
            }
          }
          else {
            DirectoryVersion* dir_version;
            FileVersion* file_version;
            int bloque;

            if (it_listA->getType() == 'd') {
              if(!getDirVersion(&dir_version, a_DirName + "//" + fname, a_VersionA))
                break;

              cout << espacio + "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                  << ", usuario: " << dir_version->getUser() << endl;

              delete dir_version;
            }
            else {
              if(!getFileVersionAndBlock(&bloque, &file_version, a_DirName + "//" + fname, a_VersionB))
                break;
              cout << espacio + "    " << fname << " version: " << file_version->getNroVersion() << ", tipo: " <<
file_version->getTipo()
                   << ", usuario: " << file_version->getUser() << endl;

              delete file_version;
            }
          }
        }
      }

      if (!found) {
        if(it_listA->getType() == 'd') {
          DirectoryVersion* dir_version;
          if(!getDirVersion(&dir_version, a_DirName + "//" + fname, a_VersionA))
            break;

          cout << espacio + "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
              << ", usuario: " << dir_version->getUser() << "-> borrado" << endl;

          delete dir_version;
        }
        else {
          FileVersion* file_version;
          int bloque;

          if (!getFileVersionAndBlock(&bloque, &file_version, a_DirName + "//" + fname, a_VersionA))
            break;

          cout << espacio + "    " << fname << " version: " << file_version->getNroVersion() << ", tipo: " <<
file_version->getTipo()
               << ", usuario: " << file_version->getUser() << "-> borrado" << endl;

          delete file_version;
        }
      }
    }
```

```
   for (it_listB = lstB->begin(); it_listB != lstB->end(); ++it_listB) {
      bool found = false;
      for(it_listA = lstA->begin(); it_listA != lstA->end(); ++it_listA) {
         if(strcmp(it_listA->getName(),it_listB->getName()) == 0)
            found = true;
      }

      if (!found) {
         fname = it_listB->getName();

         if (it_listB->getType() == 'd') {
            DirectoryVersion* dir_version;
            if(!getDirVersion(&dir_version, a_DirName + "//" + fname, a_VersionB))
               break;

            cout << espacio + "   " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                 << ", usuario: " << dir_version->getUser() << "-> agregado" << endl;

            showAddedDirectory(dir_version, a_DirName + "//" + fname, tabs + 1);
            delete dir_version;
         }
         else {
            FileVersion* file_version;
            int bloque;

            if(!getFileVersionAndBlock(&bloque, &file_version, a_DirName + "//" + fname, a_VersionB))
               break;

            cout << espacio + "   " << fname << " version: " << file_version->getNroVersion() << ", tipo: " <<
file_version->getTipo()
                 << ", usuario: " << file_version->getUser() << "-> agregado" << endl;

            delete file_version;
         }
      }
   }

   delete versionDirectorioA;
   delete versionDirectorioB;
   return ret;
}

bool VersionManager::getDiff(std::ostream& os, const string& a_VersionA, const string& a_VersionB, const string&
a_Target, const string& repositoryName)
{
   if (!_isOpen)
      return false;

   string searchingPath = repositoryName;

   // voy agregando componente a componente para saber donde debo buscar
   for (int i = 1; i < countComponents(a_Target); ++i)
      searchingPath = searchingPath + "//" + getComponent(a_Target,i);

   DirectoryVersion* versionDirectorioA;
   DirectoryVersion* versionDirectorioB;

   if (!getDirVersion(&versionDirectorioA, searchingPath,a_VersionA))
      return false;
```

```
    if (!getDirVersion(&versionDirectorioB,searchingPath,a_VersionB)) {
        delete versionDirectorioA;
        return false;
    }

    bool ret = false;

    if (a_Target != "") {
        if (versionDirectorioA->getType() == DirectoryVersion::BORRADO) {
            delete versionDirectorioA;
            delete versionDirectorioB;
            os << "No existe la version: " << a_VersionA << " de " << searchingPath << endl;
            return false;
        }

        if (versionDirectorioB->getType() == DirectoryVersion::BORRADO) {
            delete versionDirectorioA;
            delete versionDirectorioB;
            os << "No existe la version: " << a_VersionB << " de " << searchingPath << endl;
            return false;
        }

        string filename = getComponent(a_Target,countComponents(a_Target));

        File* fileA;
        if(!versionDirectorioA->searchFile(filename.c_str(),&fileA)) {
            delete versionDirectorioA;
            delete versionDirectorioB;
            os << "No existe la version: " << a_VersionA << " de " << searchingPath << "//" << filename << endl;
            return false;
        }

        File* fileB;
        if(!versionDirectorioB->searchFile(filename.c_str(),&fileB)) {
            delete fileA;
            delete versionDirectorioA;
            delete versionDirectorioB;
            os << "No existe la version: " << a_VersionB << " de " << searchingPath << "//" << filename << endl;
            return false;
        }

        if(fileA->getType() != fileB->getType()) {
            delete fileA;
            delete fileB;
            delete versionDirectorioA;
            delete versionDirectorioB;
            os << "Las versiones " << a_VersionA << " y " << a_VersionB << " de " << filename << " son tipos de archivos
diferentes" << endl;
            return false;
        }

        if(fileA->getType() != 'd')
            ret = getFileDiff(os, a_VersionA, a_VersionB,searchingPath + "//" + filename);

        else
            ret = getDirectoryDiff(searchingPath + "//" + filename, a_VersionA, a_VersionB,0);

        delete versionDirectorioA;
        delete versionDirectorioB;
```

```
        delete fileA;
        delete fileB;

        return ret;
    }
    else {
        list<File>* lstA = versionDirectorioA->getFilesList();
        list<File>* lstB = versionDirectorioB->getFilesList();

        list<File>::iterator it_listA;
        list<File>::iterator it_listB;

        string fname;

        ret = true;
        os << repositoryName << endl;

        for (it_listA = lstA->begin(); it_listA != lstA->end(); ++it_listA) {
            bool found = false;
            for(it_listB = lstB->begin(); it_listB != lstB->end(); ++it_listB) {
                if(strcmp(it_listA->getName(),it_listB->getName()) == 0) {
                    found = true;
                    fname = it_listA->getName();

                    if(it_listA->getVersion() != it_listB->getVersion()) {

                        if((it_listA->getType() != 'd') && (it_listB->getType() != 'd')) {
                            FileVersion* file_versionA;
                            FileVersion* file_versionB;

                            int bloqueA;
                            int bloqueB;

                            if(!getFileVersionAndBlock(&bloqueA, &file_versionA, searchingPath + "//" + fname, a_VersionA))
                                break;

                            if(!getFileVersionAndBlock(&bloqueB, &file_versionB, searchingPath + "//" + fname, a_VersionB)) {
                                delete file_versionA;
                                break;
                            }

                            os << "    " << fname << " version: " << file_versionA->getNroVersion() << ", tipo: " <<
file_versionA->getTipo()
                                << ", usuario: " << file_versionA->getUser() << "    ";
                            os << "    " << fname << " version: " << file_versionB->getNroVersion() << ", tipo: " <<
file_versionB->getTipo()
                                << ", usuario: " << file_versionB->getUser() << endl;

                            delete file_versionA;
                            delete file_versionB;
                        }

                        else if( (it_listA->getType() == 'd') && (it_listB->getType() == 'd') )
                            ret = ret && getDirectoryDiff(searchingPath + "//" + fname, a_VersionA, a_VersionB,1);

                        else if ( (it_listA->getType() == 'd') && (it_listB->getType() != 'd') ) {
                            DirectoryVersion* dir_version;
                            FileVersion* file_version;
                            int bloque;
```

```
                    if(!getDirVersion(&dir_version, searchingPath + "//" + fname, a_VersionA))
                        break;

                    if(!getFileVersionAndBlock(&bloque, &file_version, searchingPath + "//" + fname, a_VersionB)) {
                        delete dir_version;
                        break;
                    }

                    os << "   " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                        << ", usuario: " << dir_version->getUser() << "   ";
                    os << "   " << fname << " version: " << file_version->getNroVersion() << ", tipo: " << file_version-
>getTipo()
                        << ", usuario: " << file_version->getUser() << endl;

                    delete file_version;
                    delete dir_version;
                }
                else {
                    DirectoryVersion* dir_version;
                    FileVersion* file_version;
                    int bloque;

                    if (!getDirVersion(&dir_version, searchingPath + "//" + fname, a_VersionB))
                        break;

                    if (!getFileVersionAndBlock(&bloque, &file_version, searchingPath + "//" + fname, a_VersionA)) {
                        delete file_version;
                        break;
                    }

                    os << "   " << fname << ", version: " << file_version->getNroVersion() << ", tipo: " << file_version-
>getTipo()
                        << " usuario: " << file_version->getUser() << "   ";
                    os << "   " << fname << ", version: " << dir_version->getNroVersion() << ", tipo: d"
                        << " usuario: " << dir_version->getUser() << endl;

                    delete file_version;
                    delete dir_version;
                }
            }
            else {
                DirectoryVersion* dir_version;
                FileVersion* file_version;
                int bloque;

                if (it_listA->getType() == 'd') {
                    if(!getDirVersion(&dir_version, searchingPath + "//" + fname, a_VersionA))
                        break;

                    os << "   " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                        << ", usuario: " << dir_version->getUser() << endl;

                    delete dir_version;
                }
                else {
                    if(!getFileVersionAndBlock(&bloque, &file_version, searchingPath + "//" + fname, a_VersionB))
                        break;
                    os << "   " << fname << " version: " << file_version->getNroVersion() << ", tipo: " << file_version-
>getTipo()
                        << ", usuario: " << file_version->getUser() << endl;
```

```
                    delete file_version;
                }
            }
        }
    }

    if (!found) {
        if(it_listA->getType() == 'd') {
            DirectoryVersion* dir_version;
            if(!getDirVersion(&dir_version, searchingPath + "//" + fname, a_VersionA))
                break;

            os << "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                << ", usuario: " << dir_version->getUser() << "-> borrado" << endl;

            delete dir_version;
        }
        else {
            FileVersion* file_version;
            int bloque;

            if(!getFileVersionAndBlock(&bloque, &file_version, searchingPath + "//" + fname, a_VersionA))
                break;

            os << "    " << fname << " version: " << file_version->getNroVersion() << " tipo: " << file_version-
>getTipo() <<
                " usuario: " << file_version->getUser() << " borrado" << endl;

            delete file_version;
        }
    }
}

for (it_listB = lstB->begin(); it_listB != lstB->end(); ++it_listB) {
    bool found = false;
    for(it_listA = lstA->begin(); it_listA != lstA->end(); ++it_listA) {
        if(strcmp(it_listA->getName(),it_listB->getName()) == 0)
            found = true;
    }

    if (!found) {
        fname = it_listB->getName();

        if(it_listB->getType() == 'd') {
            DirectoryVersion* dir_version;
            if(!getDirVersion(&dir_version, searchingPath + "//" + fname, a_VersionB))
                break;

            os << "    " << fname << " version: " << dir_version->getNroVersion() << ", tipo: d"
                << " usuario: " << dir_version->getUser() << "-> agregado" << endl;

            showAddedDirectory(dir_version, searchingPath + "//" + fname, 1);
            delete dir_version;
        }
        else {
            FileVersion* file_version;
            int bloque;

            if(!getFileVersionAndBlock(&bloque, &file_version, searchingPath + "//" + fname, a_VersionB))
```

```
                    break;

                os << "    " << fname << " version: " << file_version->getNroVersion() << ", tipo: " << file_version->getTipo()
                    << ", usuario: " << file_version->getUser() << "-> agregado" << endl;

                delete file_version;
            }
          }
        }

        delete versionDirectorioA;
        delete versionDirectorioB;
        return ret;
    }

    return false;
}


bool VersionManager::getDiffByDate(std::ostream& os, const string& a_Date)
{
  if(!_isOpen)
    return false;

  int offset = _dateIndex.search(a_Date.c_str());
  if(offset < 0)
    return false;

  return _dateLog.showDate(a_Date,offset);
}

bool VersionManager::getHistory(std::ostream& os, const string& a_Filename)
{
    if (!_isOpen)
        return false;

    string searchingPath = _repository;

    if(!a_Filename.empty()) {
        for(int i = 1; i <= countComponents(a_Filename); ++i)
            searchingPath += "//" + getComponent(a_Filename, i);

        int bloque = _dirIndex.getFirstBlock(searchingPath.c_str());

        if (bloque >= 0) {
            cout << a_Filename << endl;
            _dirVersions.getHistory(os, bloque);
            return true;
        }
        else {
            bloque = _fileIndex.getFirstBlock(searchingPath.c_str());
            if (bloque < 0)
                return false;

            cout << a_Filename << endl;
            _fileVersions.getHistory(os, bloque);
            return true;
        }
    }
```

```
    else {
        int bloque = _dirIndex.search(_repository.c_str());
        int lastVersion = _dirVersions.getLastVersionNumber(bloque);
        DirectoryVersion* dv;
        if (getDirVersion(&dv, _repository, toString<int>(lastVersion))) {
            cout << _repository << endl;
            showDirectory(dv, _repository, 0);
            delete dv;
            return true;
        }
    }
    return false;
}

void VersionManager::showAddedDirectory(DirectoryVersion* dirVersion, const string& path, int tabs)
{
    string espacio;
    for (int i = 1; i <= tabs; ++i)
        espacio = espacio + "    ";

    list<File>* filesLst = dirVersion->getFilesList();
    list<File>::iterator it_files;

    for (it_files = filesLst->begin(); it_files != filesLst->end(); ++it_files) {
        string fname = it_files->getName();

        if (it_files->getType() != 'd') {
            FileVersion* fileVersion;
            int bloque;

            if (!getFileVersionAndBlock(&bloque, &fileVersion, path + "//" + fname,toString<int>(it_files->getVersion())
) )
                break;

            cout << espacio + "    " << fname << " version: " << fileVersion->getNroVersion() << ", tipo: " << fileVersion-
>getTipo() << ", usuario: "
                 << fileVersion->getUser() << "-> agregado" << endl;

            delete fileVersion;
        }
        else {
            DirectoryVersion* dirVersion;
            if(!getDirVersion(&dirVersion, path + "//" + fname, toString<int>(it_files->getVersion()) ) )
                break;

            cout << espacio + "    " << fname << " version: " << dirVersion->getNroVersion() << ", tipo: " << "d, usuario: "
<< dirVersion->getUser()
                 << "-> agregado" << endl;

            showAddedDirectory(dirVersion,path + "//" + fname, tabs + 1);
            delete dirVersion;
        }
    }
}

void VersionManager::showDirectory(DirectoryVersion* dirVersion, const string& path, int tabs)
{
    string espacio;
    for (int i = 1; i <= tabs; ++i)
        espacio = espacio + "    ";
```

```
    list<File>* filesLst = dirVersion->getFilesList();
    list<File>::iterator it_files;

    for (it_files = filesLst->begin(); it_files != filesLst->end(); ++it_files) {
        string fname = it_files->getName();

        if (it_files->getType() != 'd') {
            FileVersion* fileVersion;
            int bloque;

            if (!getFileVersionAndBlock(&bloque, &fileVersion, path + "//" + fname,toString<int>(it_files->getVersion())
 ) )
                break;

            cout << espacio + "    " << fname << " version: " << fileVersion->getNroVersion() << ", tipo: " << fileVersion-
>getTipo() << ", usuario: "
                 << fileVersion->getUser() << endl;

            delete fileVersion;
        }
        else {
            DirectoryVersion* dirVersion;
            if(!getDirVersion(&dirVersion, path + "//" + fname, toString<int>(it_files->getVersion()) ) )
                break;

            cout << espacio + "    " << fname << " version: " << dirVersion->getNroVersion() << ", tipo: " << "d, usuario: "
<< dirVersion->getUser()
                 << endl;

            showDirectory(dirVersion,path + "//" + fname, tabs + 1);
            delete dirVersion;
        }
    }
}

void VersionManager::log(const string& a_Filename, const string& a_Username, const string& a_Version, time_t
a_Date)
{
  tm* date = localtime(&a_Date);
  int anio = date->tm_year + 1900;
  int mes = date->tm_mon;
  int dia = date->tm_mday;

  string fecha = toString<int>(anio) + "/" + zeroPad(mes + 1, 2) + "/" + zeroPad(dia, 2) + " - "
               + toString<int>(date->tm_hour) + ":" + toString<int>(date->tm_min) + ":" + toString<int>(date->tm_sec);

  int offset = _dateLog.append(a_Username, fecha, a_Version, a_Filename);

  // indexo la fecha en el indice por fechas
  if (_dateIndex.search(fecha.c_str()) < 0)
    _dateIndex.insert(fecha.c_str(),offset);

  // indexo el usuario en el indice por usuario
  int ref = _usersIndex.search(a_Username.c_str());
  int nuevoBloque;
  if(ref < 0) {
    _usersReg.insertRef(offset, &nuevoBloque);
    _usersIndex.insert((a_Username + zeroPad(nuevoBloque,VERSION_DIGITS)).c_str(),nuevoBloque);
```

```
      return;
    }

    UsersRegisterFile::t_status status = _usersReg.insertRef(offset, ref, &nuevoBloque);
    switch (status) {
      case UsersRegisterFile::OK :
        return;
        break;
       case UsersRegisterFile::OVERFLOW :
        _usersIndex.insert((a_Username + zeroPad(nuevoBloque, VERSION_DIGITS)).c_str(), nuevoBloque);
        return;
        break;

       default:
          return;
          break;
     }

    return;
}


bool VersionManager::getListOfChanges(std::ostream& os, const string& a_Username, int a_Num)
{
    if (!a_Username.empty()) {
        int ref = _usersIndex.search(a_Username.c_str());
        if (ref < 0)
            return true; // no changes for user a_Username

        list<int> lstChanges;
        if (a_Num > 0)
            lstChanges = _usersReg.getReferences(ref, a_Num);
        else
            lstChanges = _usersReg.getAllReferences(ref);

        list<int>::iterator it;
        for (it = lstChanges.begin(); it != lstChanges.end(); ++it) {
            if (!_dateLog.show(*it))
                return false;
        }
        return true;
    }
    return (_dateLog.showAll());
}
```

**Repositorio.h**

```
#ifndef REPOSITORIO_H_INCLUDED
#define REPOSITORIO_H_INCLUDED

#include "ConfigData.h"
#include "SVNException.h"
#include "User.h"
#include "VersionManager.h"

#include <list>

class Repositorio
{
public:
   static const string VERSION_FILENAME;

   Repositorio(const string& a_Almacen, const string& a_Name);
   ~Repositorio() {};

   bool create();
   bool destroy();

   bool open();
   bool close();

   bool add(const string& a_Target, const string& a_Username, const string& a_Password);
   bool removeFileOrDirectory(const string& a_Target, const string& a_Username, const string& a_Password);
   bool get(const string& a_TargetDestiny, const string& a_Target, const string& a_Version,
            const string& a_Username, const string& a_Password);

   bool addUser   (const string& a_Username, const string& a_Password, const string& a_Fullname);
   bool removeUser(const string& a_Username);
   bool userExists(const string& a_Username) const;
   bool validateUser    (const string& a_Username, const string& a_Password) const;
   bool validatePassword(const string& a_Username, const string& a_Password) const;
   bool getDiff(std::ostream& os, const string& a_Username, const string& a_Password, const string& a_VersionA,
const string& a_VersionB, const string& a_Filename = "");
   bool getDiffByDate(std::ostream& os, const string& a_Username, const string& a_Password, const string& a_Date);
   bool getHistory(std::ostream& os, const string& a_Username, const string& a_Password, const string& a_Filename);
   bool getListOfChanges(std::ostream& os, const string& a_Username, const string& a_Password, int a_Num, bool
isAdmin);

   // getters
   string getName() const { return _name; }
   std::list<User> getListOfUsers() const { return _lUsers; }

protected:
   bool saveVersion();
   bool loadVersion();

private:
   Repositorio();

   // member variables
   int            _version;
   string         _name;
   string         _almacen;
   std::list<User> _lUsers;
   VersionManager  _versionManager;
```

```
    bool            _isOpen;
};

#endif
```

**Repositorio.cpp**

```
#include "Repositorio.h"
#include "debug.h"

#include <ctime>
#include <fstream>
#include <sys/stat.h>

const string Repositorio::VERSION_FILENAME = "version";

// constructor
Repositorio::Repositorio(const string& a_Almacen, const string& a_Name) :
        _version(0), _name(a_Name), _almacen(a_Almacen), _versionManager(a_Almacen, a_Name), _isOpen(false)
{
}

bool Repositorio::validateUser(const string& a_Username, const string& a_Password) const
{
    return (userExists(a_Username) && validatePassword(a_Username, a_Password));
}

bool Repositorio::validatePassword(const string& a_Username, const string& a_Password) const
{
    std::list<User>::const_iterator it;
    for (it = _lUsers.begin(); it != _lUsers.end(); ++it) {
        if (it->username == a_Username)
            return (it->password == a_Password);
    }
    return false;
}

bool Repositorio::userExists(const string& a_Username) const
{
    std::list<User>::const_iterator it;
    for (it = _lUsers.begin(); it != _lUsers.end(); ++it) {
        if (it->username == a_Username)
            return true;
    }
    return false;
}

bool Repositorio::removeFileOrDirectory(const string& a_Target, const string& a_Username, const string&
a_Password)
{
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password)) return false;

    time_t date;
    time(&date);
    if (!_versionManager.open())
        return false;
    if (!_versionManager.removeFileOrDirectory(_version + 1, _name, a_Target, a_Username, date))
```

```
        return false;

    if (!_versionManager.close())
        return false;

    _version++;
    return true;
}

bool Repositorio::add(const string& a_Target, const string& a_Username, const string& a_Password)
{
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password)) return false;

    t_filetype ftype = getFiletype(a_Target);

    if (ftype == INVALID)
        return false; // file not found

    time_t date;
    time(&date);
    if (!_versionManager.open())
        return false;
    if (!_versionManager.add(_version + 1, _name, a_Target, a_Username, date, ftype))
        return false;

    if (!_versionManager.close())
        return false;

    _version++;
    return true;
}

bool Repositorio::addUser(const string& a_Username, const string& a_Password, const string& a_Fullname)
{
    if (userExists(a_Username))
        return false;

    User user;
    user.username = a_Username;
    user.password = a_Password;
    user.fullname = a_Fullname;
    _lUsers.push_back(user);

    return true;
}

bool Repositorio::removeUser(const string& a_Username)
{
    std::list<User>::iterator it;
    for (it = _lUsers.begin(); it != _lUsers.end(); ++it) {
        if (it->username == a_Username) {
            return true;
        }
    }

    return false; // user not found
}
```

```
bool Repositorio::create()
{
   if (_isOpen)
      return false;

   debug("creating Repositorio '" + _name + "' in Almacen '" + _almacen + "'\n");
   // create directory where files will be stored
   mkdir((_almacen + "//" + _name).c_str(), 0755);
   _isOpen = saveVersion() && _versionManager.create();
   if (!_isOpen)
      remove(_name.c_str());
   debug("Repositorio creation " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool Repositorio::destroy()
{
   debug("destroying Repositorio '" + _name + "' in Almacen '" + _almacen + "'\n");
   if (!_isOpen)
      _isOpen = open();

   bool ret = _isOpen && _versionManager.destroy();
   // remove version file
   ret = ret && (remove((_almacen + "//" + _name + "//" + VERSION_FILENAME).c_str()) == 0);
   // remove repository directory
   ret = ret && (remove((_almacen + "//" + _name).c_str()) == 0);
   debug("Repositorio destroy " + string(ret ? "successfully" : "failed") + "\n");
   return ret;
}

bool Repositorio::open()
{
   if (_isOpen)
      return true;

   debug("opening Repositorio '" + _name + "' in Almacen '" + _almacen + "'\n");
   _isOpen = loadVersion();
   _isOpen = _isOpen && _versionManager.open();
   debug("Repositorio open " + string(_isOpen ? "successfull" : "failed") + "\n");
   return _isOpen;
}

bool Repositorio::close()
{
   if (!_isOpen)
      return true;

   debug("closing Repositorio '" + _name + "' in Almacen '" + _almacen + "'\n");
   if (!saveVersion())
      return false;
   _isOpen = !_versionManager.close();
   debug("Repositorio close " + string(!_isOpen ? "successfull" : "failed") + "\n");
   return !_isOpen;
}

bool Repositorio::saveVersion()
{
   // saves a file named "version" containing the repository version
   debug("saving repository version to " + toString(_version) + "\n");
```

```
    string filename = _almacen + "//" + _name + "//" + VERSION_FILENAME;
    std::ofstream os(filename.c_str());
    if (!os.is_open()) return false;
    os << _version << std::endl;
    os.close();
    return true;
}

bool Repositorio::loadVersion()
{
    debug("loading repository version\n");
    // saves a file named "version" containing the repository version
    string filename = _almacen + "//" + _name + "//" + VERSION_FILENAME;
    std::ifstream is(filename.c_str());
    if (!is.is_open()) return false;
    is >> _version;
    is.close();
    return true;
}

bool Repositorio::get(const string& a_TargetDestiny, const string& a_Target, const string& a_Version,
                      const string& a_Username, const string& a_Password)
{
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password))
        return false;

    return _versionManager.get(a_Version, a_Target, _name, a_TargetDestiny);
}

bool Repositorio::getDiff(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_VersionA, const string& a_VersionB, const string& a_Filename)
{
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password))
        return false;

    return _versionManager.getDiff(os, a_VersionA, a_VersionB, a_Filename, _name);
}

bool Repositorio::getDiffByDate(std::ostream& os, const string& a_Username, const string& a_Password, const
string& a_Date)
{
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password))
        return false;

    return _versionManager.getDiffByDate(os, a_Date);
}


bool Repositorio::getHistory(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_Filename)
{
```

```
    if (!_isOpen)
        return false;

    if (!validateUser(a_Username, a_Password))
        return false;

    return _versionManager.getHistory(os, a_Filename);
}


bool Repositorio::getListOfChanges(std::ostream& os, const string& a_Username, const string& a_Password, int
a_Num, bool isAdmin)
{
    if (!_isOpen)
        return false;

    if (isAdmin) {
        if (!a_Username.empty() && !userExists(a_Username))
            return false;
    }
    else if (!validateUser(a_Username, a_Password))
        return false;

    return _versionManager.getListOfChanges(os, a_Username, a_Num);
}
```

**Almacen.h**

```
#ifndef ALMACEN_H_INCLUDED
#define ALMACEN_H_INCLUDED

#include "ConfigData.h"
#include "Repositorio.h"
#include "User.h"

#include <xercesc/util/PlatformUtils.hpp>
#include <list>
#include <sstream>

using std::string;

class Almacen
{
public:
   static string CONFIG_FILE;

   Almacen(); // constructor
   ~Almacen();

   bool create         (const string& a_Name);
   bool destroy();
   bool exists() const { return _exists; }

   bool addRepository   (const string& a_Name) throw();
   bool removeRepository(const string& a_Name) throw();
   bool repositoryExists(const string& a_Name) throw();

   bool addUser(const string& a_Reposit, const string& a_Username, const string& a_Password, const string&
a_Fullname) throw();
   bool removeUser(const string& a_Reposit, const string& a_Username) throw();
   bool userExists(const string& a_Reposit, const string& a_Username) const;
   bool validatePassword(const string& a_Reposit, const string& a_Username, const string& a_Password) const;
   std::list<User>   getListOfUsers(const string& a_Reposit) const;
   bool getListOfChanges(std::ostream& os, const string& a_Reposit, const string& a_Username, const string&
a_Password, int a_Num, bool isAdmin);
   bool changePassword(const string& a_Reposit, const string& a_Username, const string& a_NewPassword);

   bool add(const string& a_Reposit, const string& a_Target, const string& a_Username, const string& a_Password);
   bool remove(const string& a_Reposit, const string& a_Target, const string& a_Username, const string&
a_Password);
   bool get(const string& a_Reposit, const string& a_TargetDestiny, const string& a_Target, const string& a_Version,
            const string& a_Username, const string& a_Password) const;
   bool getDiff(std::ostream& os, const string& a_Username, const string& a_Password, const string& a_Reposit, const
string& a_VersionA, const string& a_VersionB, const string& a_Filename = "");
   bool getDiffByDate(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_Reposit, const string& a_Date);
   bool getHistory(std::ostream& os, const string& a_Username, const string& a_Password, const string& a_Reposit,
const string& a_Filename);

   bool getLock    (const string& a_Name) const;
   bool releaseLock(const string& a_Name) const;

   // getters
   string getName() const { return _name; }

   bool createConfigFile(const string& a_Dir);
```

```
private:
   bool load() throw(xercesc::XMLException&);
   Repositorio* getRepository(const string& a_Name) const;

   // member variables
   string                _name;
   XMLConfigData*        _config;
   std::list<Repositorio*> _lReposit;
   bool                  _exists;
};

#endif
```

**Almacen.cpp**

```
#include "Almacen.h"
#include "debug.h"

#include <iostream>
#include <sys/stat.h>
#include <unistd.h>

using std::cout;
using std::endl;

string Almacen::CONFIG_FILE = "";

// constructor
Almacen::Almacen() : _name(""), _config(NULL)
{
   _exists = load();
}

Almacen::~Almacen()
{
   // TODO
   //delete _config;
   //std::list<Repositorio*>::iterator it;
   //for (it = _lReposit.begin(); it != _lReposit.end(); ++it)
      //delete *it;
   debug("Destroying Almacen '" + _name + "'\n");
}

bool Almacen::createConfigFile(const string& a_Dir)
// crea el archivo de configuracion con un solo elemento:
// <almacen="...">
// </almacen>
{
   std::ofstream file(CONFIG_FILE.c_str());
   if (!file) {
      return false;
   }
   file << "<almacen dir=\"" << a_Dir << "\">" << endl
        << "</almacen>" << endl;
   file.close();
   return true;
}

bool Almacen::load() throw(xercesc::XMLException&)
```

```
{
   // load from xml file

   debug("loading Almacen\n");
   bool ret = true;
   try {
      // initialize xerces
      xercesc::XMLPlatformUtils::Initialize();

      CONFIG_FILE = "//home//" + string(getenv("USER")) + "//.svn_grupo_config";
      _config = new XMLConfigData(CONFIG_FILE);
      _name = _config->getDirAlmacen();

      // typedef vector<pair<string, UsersList> > RepositoriosList;
      XMLConfigData::RepositoriosList replist = _config->getRepositories();

      XMLConfigData::RepositoriosList::iterator repIt;
      for (repIt = replist.begin(); repIt != replist.end(); ++repIt) {
         Repositorio* rep = new Repositorio(_name, repIt->first);
         XMLConfigData::UsersList::iterator usIt;
         for (usIt = repIt->second.begin(); usIt != repIt->second.end(); ++usIt) {
            ret = ret && rep->addUser(usIt->username, usIt->password, usIt->fullname);
         }
         if (ret)
            _lReposit.push_back(rep);
      }

      // terminate xerces
      xercesc::XMLPlatformUtils::Terminate();
   }
   catch (...) {
      debug("catched exception!\n");
      ret = false;
   }

   debug("Almacen load " + string(ret ? "successfull" : "failed") + "\n");
   return ret;
}

bool Almacen::destroy()
{
   _exists = false;
   string cmd = "rm -rf " + systemFilename(_name);
   // remove dir Almacen
   bool ok = (system(cmd.c_str()) != -1);
   // remove config file
   ok = ok && (std::remove(CONFIG_FILE.c_str()) == 0);
   return ok;
}

bool Almacen::addRepository(const string& a_Name) throw()
{
   if (!_exists)
      return false;

   try {
      Repositorio* rep = new Repositorio(_name, a_Name);
      if (!rep->create())
         return false;
```

```
        _lReposit.push_back(rep);

        if(!rep->close())
            return false;

        // add to config file
        // initialize xerces
        xercesc::XMLPlatformUtils::Initialize();

        if (!_config->addRepository(a_Name))
            return false;
        _config->commit();

        // terminate xerces
        xercesc::XMLPlatformUtils::Terminate();
    }
    catch (...) {
        return false;
    }

    return true;
}

bool Almacen::create(const string& a_Name)
{
    if (_exists) return false;

    // create configuration file
    if (_exists = createConfigFile(a_Name)) {
        // create directory where files will be stored
        mkdir(a_Name.c_str(), 0755);
        return true;
    }
    return false;
}

Repositorio* Almacen::getRepository(const string& a_Name) const
{
    std::list<Repositorio*>::const_iterator it;
    for (it = _lReposit.begin(); it != _lReposit.end(); ++it) {
        if ((*it)->getName() == a_Name)
            return *it;
    }
    return NULL;
}


bool Almacen::removeRepository(const string& a_Name) throw()
{
    if (!_exists)
        return false;

    Repositorio* rep = getRepository(a_Name);
    if (rep == NULL)
        return false;

    if (!rep->destroy())
        return false;

    _lReposit.remove(rep);
```

```
    try {
        // initialize xerces
        xercesc::XMLPlatformUtils::Initialize();

        if (!_config->removeRepository(a_Name))
            return false;
        _config->commit();

        // terminate xerces
        xercesc::XMLPlatformUtils::Terminate();
    }
    catch (...) {
        return false;
    }

    return true;
}

bool Almacen::addUser(const string& a_Reposit, const string& a_Username, const string& a_Password, const string&
a_Fullname) throw()
{
    Repositorio* rep = getRepository(a_Reposit);
    if (rep == NULL)
        return false;

    bool ret = rep->addUser(a_Username, a_Password, a_Fullname);

    if (ret)
        try {
            // initialize xerces
            xercesc::XMLPlatformUtils::Initialize();

            if (!_config->addUser(a_Reposit, a_Username, a_Password, a_Fullname))
                return false;
            _config->commit();

            // terminate xerces
            xercesc::XMLPlatformUtils::Terminate();
        }
        catch (...) {
            return false;
        }

    return ret;
}

bool Almacen::removeUser(const string& a_Reposit, const string& a_Username) throw()
{
    Repositorio* rep = getRepository(a_Reposit);
    if (rep == NULL)
        return false;

    bool ret = rep->removeUser(a_Username);

    if (ret)
        try {
            // initialize xerces
            xercesc::XMLPlatformUtils::Initialize();
```

```
            if (!_config->removeUser(a_Reposit, a_Username))
               return false;
            _config->commit();

            // terminate xerces
            xercesc::XMLPlatformUtils::Terminate();
         }
         catch (...) {
            return false;
         }

   return ret;
}

bool Almacen::add(const string& a_Reposit, const string& a_Target, const string& a_Username, const string&
a_Password)
{
   const int MAX_WAIT = 1000;
   int ntries = 0;

   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   bool ret = false;
   if (rep->open()) {
      while (!getLock(a_Target) && (ntries < MAX_WAIT))
         ntries++;                    // perform active wait, could be improved!
      if (ntries == MAX_WAIT)
         return false;       // could not get lock!
      ret = rep->add(a_Target, a_Username, a_Password);
      releaseLock(a_Target);
      rep->close();
   }
   return ret;
}

bool Almacen::remove(const string& a_Reposit, const string& a_Target, const string& a_Username, const string&
a_Password)
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;
   return rep->removeFileOrDirectory(a_Target, a_Username, a_Password);
}

bool Almacen::validatePassword(const string& a_Reposit, const string& a_Username, const string& a_Password) const
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;
   return rep->validatePassword(a_Username, a_Password);
}

bool Almacen::userExists(const string& a_Reposit, const string& a_Username) const
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;
   return rep->userExists(a_Username);
```

```
}

bool Almacen::repositoryExists(const string& a_Name) throw()
{
   return (getRepository(a_Name) != NULL);
}


std::list<User> Almacen::getListOfUsers(const string& a_Reposit) const
{
   std::list<User> users;
   Repositorio* rep = getRepository(a_Reposit);
   if (rep != NULL) {
      users = rep->getListOfUsers();
   }
   return users;
}

bool Almacen::get(const string& a_Reposit, const string& a_TargetDestiny, const string& a_Target, const string&
a_Version,
         const string& a_Username, const string& a_Password) const
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   bool ret = rep->open();
   ret = ret && rep->get(a_TargetDestiny, a_Target, a_Version, a_Username, a_Password);
   ret = ret && rep->close();
   return ret;
}

bool Almacen::getDiff(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_Reposit, const string& a_VersionA, const string& a_VersionB, const string& a_Filename)
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   bool ret = rep->open();
   ret = ret && rep->getDiff(os, a_Username, a_Password, a_VersionA, a_VersionB, a_Filename);
   ret = ret && rep->close();
   return ret;
}

bool Almacen::getDiffByDate(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_Reposit, const string& a_Date)
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   bool ret = rep->open();
   ret = ret && rep->getDiffByDate(os, a_Username, a_Password, a_Date);
   ret = ret && rep->close();
   return ret;
}

bool Almacen::getHistory(std::ostream& os, const string& a_Username, const string& a_Password, const string&
a_Reposit, const string& a_Filename)
```

```
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   bool ret = rep->open();
   ret = ret && rep->getHistory(os, a_Username, a_Password, a_Filename);
   ret = ret && rep->close();
   return ret;
}

bool Almacen::changePassword(const string& a_Reposit, const string& a_Username, const string& a_NewPassword)
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;

   try {
      // initialize xerces
      xercesc::XMLPlatformUtils::Initialize();

      if (!_config->changePassword(a_Reposit, a_Username, a_NewPassword))
         return false;
      _config->commit();

      // terminate xerces
      xercesc::XMLPlatformUtils::Terminate();
   }
   catch (...) {
      return false;
   }
   return false;
}

bool Almacen::getLock(const string& a_Name) const
{
   string filename = "." + a_Name;
   std::ifstream is(filename.c_str());
   if (is.is_open())
      return false;

   std::ofstream os(filename.c_str());
   os.close();
   return true;
}

bool Almacen::releaseLock(const string& a_Name) const
{
   string filename = "." + a_Name;
   std::remove(filename.c_str());
   return true;
}

bool Almacen::getListOfChanges(std::ostream& os, const string& a_Reposit, const string& a_Username, const string&
a_Password, int a_Num, bool isAdmin)
{
   Repositorio* rep = getRepository(a_Reposit);
   if (rep == NULL)
      return false;
```

```
    bool ret = rep->open();
    ret = ret && rep->getListOfChanges(os, a_Username, a_Password, a_Num, isAdmin);
    ret = ret && rep->close();
    return ret;
}
```

**helpers.h**

```
#ifndef HELPERS_H_INCLUDED
#define HELPERS_H_INCLUDED

#include <string>
#include <sstream>

std::string zeroPad(int number, int ndigits);
std::string randomFilename(const std::string& prefix);
std::string systemFilename(const std::string& a_Filename);
bool isEmptyFile(const std::string& a_Filename);
bool areDifferentFiles(const std::string& f1, const std::string& f2);

enum t_filetype { INVALID = -1, DIRECTORY = 0, TEXT, BINARY };
t_filetype getFiletype(const std::string& filename);

template<typename T>
std::string toString(T n)
{
    std::ostringstream os;
    os << n;
    return os.str();
}

template<typename T>
T fromString(const std::string& s)
{
    std::istringstream is(s);
    T r;
    is >> r;
    return r;
}

int countComponents(const std::string& a_Target);
std::string getComponent(const std::string& a_Target, int component);

#endif
```

**helpers.cpp**

```
#include "helpers.h"

#include <cstdlib>      // rand
#include <fstream>
#include <sys/stat.h>

std::string zeroPad(int number, int ndigits)
{
    std::string ret(ndigits, '0');
    std::string num = toString<int>(number);

    int j = 0;
    for (int i = ndigits - num.length(); i < ndigits; ++i) {
```

```
        ret[i] = num[j];
        j++;
    }
    return ret;
}


std::string randomFilename(const std::string& prefix)
// returns a string with a random integer following the prefix
{
    // a call to seed with a time_t could be used
    // if not, this is quite trivial
    std::string filename(prefix);
    int i = rand();
    return filename + toString(i);
}


t_filetype getFiletype(const std::string& filename)
{
    struct stat ss;
    if (stat(filename.c_str(), &ss) == -1)
        return INVALID; // error, file does not exist

    if ((ss.st_mode & S_IFMT) == S_IFDIR)
        return DIRECTORY;

    // determine if filename is a text or binary file
    // by reading first 512 bytes and searching for chars < 30
    // text files have majority of bytes > 30 while binary files don't
    std::ifstream is(filename.c_str());
    if (!is) return INVALID; // error, could not open file

    int bytesRead = 0;
    int nChars    = 0;
    int nNonchars = 0;
    while (!is.eof() && bytesRead++ < 512) {
        char c;
        is >> c;
        (c > 30) ? nChars++ : nNonchars++;
    }
    is.close();
    return (nChars > nNonchars) ? TEXT : BINARY;
}


int countComponents(const std::string& a_Target)
{
  int cantComponentes = 0;
  unsigned int length = a_Target.length();
  unsigned int index = 0;

  while(index < length)
    {
      index = a_Target.find_first_of("/",index);
      cantComponentes++;
      if(index < length)
            index++;
    }
```

```
  return cantComponentes;
}


std::string getComponent(const std::string& a_Target,int component)
{
	int cantidadComponentes = countComponents(a_Target);
	std::string res;
	if (component > cantidadComponentes)
		res = "";
	else {
		int componenteLeida = 0;
		unsigned int comienzo = 0;
		unsigned int fin = 0;

		while(componenteLeida != component) {
			comienzo = fin;
			fin = a_Target.find_first_of("/",fin);
			componenteLeida++;

			if(fin < a_Target.length())
				fin++;
		}
		res = a_Target.substr(comienzo,fin - comienzo - 1);
	}
	return res;
}


bool isEmptyFile(const std::string& a_Filename)
{
   std::ifstream is(a_Filename.c_str());
   if (!is.is_open())
      return true;

   std::string line;
   getline(is, line);
   if (!is.eof())
      return false;
   return line.empty();
}


bool areDifferentFiles(const std::string& f1, const std::string& f2)
// returns true if f1 and f2 are different
// assumes both files exist
{
   std::string tmp = randomFilename(".tmp_");
   system(("diff " + systemFilename(f1) + " " + systemFilename(f2) + " > " + tmp).c_str());
   bool empty = !isEmptyFile(tmp);
   remove(tmp.c_str());
   return !empty;
}


std::string systemFilename(const std::string& a_Filename)
// prepends spaces in a_Filename with '\'
{
   std::string ret;
   for (int i = 0; i < (int)a_Filename.length(); ++i) {
```

```
        if (a_Filename[i] == ' ')
            ret += "\\ ";
        else
            ret += a_Filename[i];
    }
    return ret;
}
```

**SVNException.h**

```
#ifndef SVNEXCEPTION_H_INCLUDED
#define SVNEXCEPTION_H_INCLUDED

#include <exception>

class SVNException : public std::exception
{
};

#endif
```

**main-user.cpp**

```
#include "Almacen.h"
#include "debug.h"
#include "helpers.h"

#include <exception>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

using std::cout;
using std::cerr;
using std::endl;
using std::string;

bool validateUserAndRepository(Almacen* a_Almacen, const string& a_Reposit, const string& a_Username, const
string& a_Password)
{
    if (!a_Almacen->repositoryExists(a_Reposit)) {
        cout << "El repositorio " << a_Reposit << " no existe." << endl;
        return false;
    }

    if (!a_Almacen->userExists(a_Reposit, a_Username)) {
        cout << "El usuario " << a_Username << " no pertenece al repositorio " << a_Reposit << "." << endl;
        return false;
    }

    if (!a_Almacen->validatePassword(a_Reposit, a_Username, a_Password)) {
        cout << "Contraseña invalida." << endl;
        return false;
    }

    return true;
}
```

```
void add(const string& a_Username, const string& a_Password, const string& a_Reposit, const string& a_Target)
{
   Almacen almacen;
   if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
      return;

   if (!almacen.add(a_Reposit, a_Target, a_Username, a_Password)) {
      cout << "El archivo " << a_Target << " no pudo ser agregado." << endl;
      return;
   }

   cout << "El archivo " << a_Target << " ha sido agregado exitosamente." << endl;
}


void showHistory(const string& a_Username, const string& a_Password, const string& a_Reposit, const string&
a_Filename = "")
{
   Almacen almacen;
   if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
      return;

   if (!almacen.getHistory(cout, a_Username, a_Password, a_Reposit, a_Filename))
      cout << "No se pudo obtener la historia del archivo solicitado." << endl;
}


void showDiff(const string& a_Username, const string& a_Password, const string& a_Reposit, const string&
a_VersionA, const string& a_VersionB, const string& a_Filename = "")
{
   Almacen almacen;
   if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
      return;

   if (!almacen.getDiff(cout, a_Username, a_Password, a_Reposit, a_VersionA, a_VersionB, a_Filename)) {
      cout << "No se pudieron obtener las versiones solicitadas." << endl;
      return;
   }
}


void showByDate(const string& a_Username, const string& a_Password, const string& a_Reposit, const string&
a_Date)
{
   Almacen almacen;
   if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
      return;

   if (!almacen.getDiffByDate(cout, a_Username, a_Password, a_Reposit, a_Date))
      cout << "No se pudieron obtener las versiones solicitadas." << endl;
}

void changePassword(const string& a_Username, const string& a_Password, const string& a_NewPassword, const
string& a_Reposit)
{
   Almacen almacen;
   if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
      return;

   if (almacen.changePassword(a_Reposit, a_Username, a_NewPassword))
```

```
        cout << "La  contraseña ha sido actualizada exitosamente." << endl;
    else
        cout << "La  contraseña no ha podido ser actualizada." << endl;


}

void get(const string& a_Username, const string& a_Password, const string& a_Reposit, const string& a_TargetDir,
const string& a_Filename, const string& a_Version)
{
    Almacen almacen;
    if (!validateUserAndRepository(&almacen, a_Reposit, a_Username, a_Password))
        return;

    if (!almacen.get(a_Reposit, a_TargetDir, a_Filename, a_Version, a_Username, a_Password)) {
        cout << "El archivo " << a_Filename << " no pudo ser recuperado." << endl;
        return;
    }
    cout << "El archivo " << a_Filename << " ha sido recuperado exitosamente." << endl;
}


void showChanges(const string& a_Username, const string& a_Password, const string& a_Reposit, const string&
a_Num = "")
{
    Almacen almacen;
    if (!almacen.exists()) {
        cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    if (!almacen.repositoryExists(a_Reposit)) {
        cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    cout << "Lista de ultimas modificaciones al repositorio '" << a_Reposit << "':" << endl;
    if (!almacen.getListOfChanges(cout, a_Reposit, a_Username, a_Password, fromString<int>(a_Num), false)) {
        cout << "No se ha podido obtener el listado, verifique que el usuario y contraseña sean validos." << endl;
        return;
    }
}


void showHelp(const char* progname)
{
    cout << "uso: " << progname << " usuario contraseña [[-a \"nombre repositorio\" \"nombre archivo/directorio\"] |" <<
endl
        << "                [-d \"nombre repositorio\" version_inicial version_final] [\"nombre archivo/directorio\"]] |" << endl
        << "                [-f \"nombre repositorio\" fecha(aaaa/mm/dd)]] | [-h] |" << endl
        << "                [-l \"nombre repositorio\" [\"nombre archivo/directorio\"]] |" << endl
        << "                [-m \"nombre repositorio\" [cantidad]] |" << endl
        << "                [-o \"nombre repositorio\" \"nombre directorio destino\" \"nombre archivo/directorio\" [version]] |"
<< endl
        << "                [-p nueva \"nombre repositorio\" ]]" << endl
        << "-a, almacenar archivos y directorios." << endl
        << "-d, ver diferencias entre dos versiones." << endl
        << "-f, ver las actualizaciones en una fecha dada." << endl
        << "-l, ver historial de cambios a un archivo o directorio." << endl
        << "-m, obtener listado de ultimos cambios efectuados por un usuario." << endl
```

```
        << "-o, obtener una determinada version de un archivo o directorio." << endl
        << "-p, cambiar contraseña." << endl
        << endl
        << "-h, mostrar esta ayuda." << endl
        ;
}


int main(int argc, char** argv)
{
    int c;
    string user, pass;
    bool argsok = false;
    while ((c = getopt(argc, argv, "-a:d:f:hl:m:o:p:")) != -1) {
        switch (c) {

            case 1: // non-option ARGV-elements
                if (user.length() == 0)
                    user = optarg;
                else if (pass.length() == 0)
                    pass = optarg;
                // else invalid argument
                break;

            case 'a': // agregar archivo o cambios al repositorio
                // -a "nombre repositorio" "nombre archivo/directorio"
                argsok = (argc == 6);
                if (argsok) add(user, pass, optarg, argv[optind]);
                break;

            case 'd': // mostrar diff entre 2 versiones
                // -d "nombre repositorio" version_inicial version_final ["nombre archivo/directorio"]
                argsok = ((argc == 7) || (argc == 8));
                if (argsok) {
                    if (argc == 7)
                        showDiff(user, pass, optarg, argv[optind], argv[optind + 1], "");
                    else
                        showDiff(user, pass, optarg, argv[optind], argv[optind + 1], argv[optind + 2]);
                }
                break;

            case 'f': // mostrar cambios en determinada fecha
                // -f usuario "nombre repositorio" fecha
                argsok = (argc == 6);
                if (argsok) showByDate(user, pass, optarg, argv[optind]);
                break;

            case 'h':
                showHelp(argv[0]);
                argsok = true;
                break;

            case 'l': // cambios a un archivo
                // -l "nombre repositorio" "nombre archivo/dir"
                argsok = ((argc == 5) || (argc == 6));
                if (argsok)
                    if (argc == 5)
                        showHistory(user, pass, optarg, "");
                    else
                        showHistory(user, pass, optarg, argv[optind]);
```

```
            break;

        case 'm': // listar cambios de usuario
            // -m rep [cant]
            argsok = ((argc == 5) || (argc == 6));
            if (argsok) {
                if (argc == 5)
                    showChanges(user, pass, optarg);
                else
                    showChanges(user, pass, optarg, argv[optind]);
            }
            break;

        case 'o': // obtener archivo
            // -o "nombre repositorio" "nombre directorio destino" ["nombre archivo/dir"] [version]
            argsok = ((argc == 6) || (argc == 7) || (argc == 8));
            if (argsok) {
                if (argc == 6)
                    get(user, pass, optarg, argv[optind], "", "");
                else if (argc == 7)
                    get(user, pass, optarg, argv[optind], argv[optind + 1], "");
                else
                    get(user, pass, optarg, argv[optind], argv[optind + 1], argv[optind + 2]);
            }
            break;

        case 'p': // cambiar password
            // -p nueva "nombre repositorio"
            argsok = (argc == 6);
            if (argsok) changePassword(user, pass, optarg, argv[optind]);
            break;
      }
   }

   if (!argsok) {
     cout << "Parametros invalidos." << endl;
     showHelp(argv[0]);
   }

}
```

**main-admin.cpp**

```
#include <iostream>
#include <cstdlib>      // getenv
#include <fstream>
#include <string>

#include "Almacen.h"
#include "debug.h"
#include "User.h"

using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::string;

string CONFIG_FILE;
```

```
void addRepository(const string& a_Name)
{
   Almacen almacen;
   if (!almacen.exists()) {
      cout << "No existe un almacen para alojar el repositorio." << endl;
      return;
   }

   // agregar repositorio al archivo de configuracion
   if (!almacen.addRepository(a_Name)) {
      cout << "No se ha creado el repositorio '" << a_Name << "'." << endl
           << "Verifique que no exista otro con el mismo nombre." << endl;
      return;
   }

   cout << "Se ha creado el repositorio '" << a_Name << "' exitosamente." << endl;
}


void createAlmacen(const string& a_Dir)
{
   Almacen almacen;
   if (almacen.exists()) {
      char r;
      do {
         cout << "Ya existe un almacen. Desea removerlo y crear uno nuevo? (s/n): ";
         cin >> r;
      } while (r != 'n' && r != 'N' && r != 's' && r != 'S');
      if (r == 's' || r == 'S')
         almacen.destroy();
      else
         return;
   }

   if (almacen.create(a_Dir))
      cout << "El almacen '" << a_Dir << "' ha sido creado con exito." << endl;
   else
      cout << "El almacen '" << a_Dir << "' no ha sido creado." << endl;
}


void removeAlmacen()
{
   Almacen almacen;
   if (almacen.exists()) {
      char r;
      do {
         cout << "Esta seguro que desea remover el almacen '" << almacen.getName() << "'? (s/n): ";
         cin >> r;
      } while (r != 'n' && r != 'N' && r != 's' && r != 'S');
      if (r == 's' || r == 'S') {
         if (almacen.destroy());
            cout << "El almacen '" << almacen.getName() << "' ha sido removido con exito." << endl;
      }
   }
   else
      cout << "No existe ningun almacen de repositorios." << endl;
}
```

```
void addUser(const string& a_Username, const string& a_Password, const string& a_Fullname, const string&
a_Reposit)
{
   Almacen almacen;
   if (!almacen.exists()) {
      cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
      return;
   }

   if (!almacen.repositoryExists(a_Reposit)) {
      cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
      return;
   }

   if (!almacen.addUser(a_Reposit, a_Username, a_Password, a_Fullname)) {
      cout << "El usuario '" << a_Username << "' no ha sido creado." << endl
           << "Verifique que el usuario no pertenezca ya a dicho repositorio." << endl;
      return;
   }

   cout << "El usuario " << a_Username << " ha sido creado exitosamente." << endl;
}


bool fileExists(const string& a_Filename)
{
   std::ifstream file(a_Filename.c_str());
   return file.good();
}


void removeRepository(const string& a_Reposit)
{
   Almacen almacen;
   if (!almacen.exists()) {
      cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
      return;
   }

   if (!almacen.repositoryExists(a_Reposit)) {
      cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
      return;
   }

   if (almacen.removeRepository(a_Reposit))
      cout << "El repositorio '" << a_Reposit << "' ha sido eliminado exitosamente." << endl;
   else
      cout << "El repositorio '" << a_Reposit << "' no ha sido eliminado." << endl;
}


void removeUser(const string& a_Reposit, const string& a_Username)
{
   Almacen almacen;
   if (!almacen.exists()) {
      cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
      return;
   }

   if (!almacen.repositoryExists(a_Reposit)) {
```

```
        cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    if (almacen.removeUser(a_Reposit, a_Username))
        cout << "El usuario " << a_Username << " ha sido eliminado del repositorio '" << a_Reposit << "' exitosamente."
<< endl;
    else
        cout << "El usuario " << a_Username << " ha sido eliminado del repositorio '" << a_Reposit << "'." << endl;
}


void showHelp(const char* progname)
{
    cout << "uso: " << progname << " [[-a \"nombre almacen\"] | [-c \"nombre repositorio\"] | [-d] |" << endl
         << "                [-e usuario \"nombre repositorio\"] | [-h] | " << endl
         << "                [-m \"nombre repositorio\" [[\"nombre usuario\"] [cantidad]]] |" << endl
         << "                [-o \"nombre repositorio\"] |" << endl
         << "                [-r \"nombre repositorio\"] |" << endl
         << "                [-u usuario password \"nombre usuario\" \"nombre repositorio\"]]" << endl
         << "-a, crear almacen de repositorios." << endl
         << "-c, crear repositorio." << endl
         << "-d, eliminar almacen de repositorios." << endl
         << "-e, eliminar usuario." << endl
         << "-m, obtener listado de ultimos cambios efectuados por un usuario." << endl
         << "-o, obtener listado de usuarios." << endl
         << "-r, eliminar repositorio." << endl
         << "-u, agregar usuario." << endl
         << endl
         << "-h, mostrar esta ayuda." << endl
         ;
}


void showUsers(const string& a_Reposit)
{
    Almacen almacen;
    if (!almacen.exists()) {
        cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    if (!almacen.repositoryExists(a_Reposit)) {
        cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    std::list<User> users = almacen.getListOfUsers(a_Reposit);
    cout << "Lista de usuarios del repositorio '" << a_Reposit << "':" << endl
         << "[usuario] [nombre completo]" << endl;
    for (std::list<User>::iterator it = users.begin(); it != users.end(); ++it) {
        cout << it->username << " '" << it->fullname << "'" << endl;
    }
}


void showChanges(const string& a_Reposit, const string& a_Username, const string& a_Num = "")
{
    Almacen almacen;
    if (!almacen.exists()) {
```

```
        cout << "No existe un almacen que contenga el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    if (!almacen.repositoryExists(a_Reposit)) {
        cout << "No existe el repositorio '" << a_Reposit << "'" << endl;
        return;
    }

    cout << "Lista de ultimas modificaciones al repositorio '" << a_Reposit << "':" << endl;
    almacen.getListOfChanges(cout, a_Reposit, a_Username, "", fromString<int>(a_Num), true);
    //cout << is.str() << endl;
}


int main(int argc, char** argv)
{
    int c;
    bool argsok = false;
    while ((c = getopt(argc, argv, "-a:c:de:hm:o:r:u:")) != -1) {
        switch (c) {
            case 'a': // crear almacen de repositorios
                // -a "nombre directorio"
                argsok = (argc == 3);
                if (argsok) createAlmacen(optarg);
                break;

            case 'c': // agregar repositorio
                // -c "nombre repositorio"
                argsok = (argc == 3);
                if (argsok) addRepository(optarg);
                break;

            case 'd': // eliminar almacen
                argsok = true;
                removeAlmacen();
                break;

            case 'e': // eliminar usuario
                // -e usuario "nombre repositorio"
                argsok = (argc == 4);
                if (argsok) removeUser(argv[optind], optarg);
                break;

            case 'h': // mostrar ayuda
                showHelp(argv[0]);
                argsok = true;
                break;

            case 'm': // listar cambios de usuario
                // -m rep [["usuario"] [cant]]
                argsok = ((argc == 3) || (argc == 4) || (argc == 5));
                if (argsok) {
                    if (argc == 3)
                        showChanges(optarg, "");
                    else if (argc == 4)
                        showChanges(optarg, argv[optind]);
                    else
                        showChanges(optarg, argv[optind], argv[optind + 1]);
                }
```

```
            break;

        case 'o': // obtener listado de usuarios
            // -o "nombre repositorio"
            argsok = (argc == 3);
            if (argsok) showUsers(optarg);
            break;

        case 'r': // eliminar repositorio
            // -r "nombre repositorio"
            argsok = (argc == 3);
            if (argsok) removeRepository(optarg);
            break;

        case 'u': // crear usuario
            // -u usuario password "nombre usuario" "nombre repositorio"
            argsok = (argc == 6);
            if (argsok) addUser(optarg, argv[optind], argv[optind + 1], argv[optind + 2]);
            break;
      }
   }

   if (!argsok) {
     cout << "Parametros invalidos." << endl;
     showHelp(argv[0]);
   }
   return 0;
}
```