Práctica 1 - Organización de ficheros

- **Nombre**: Quintín Mesa Romero
- **Fecha**: 22 de marzo de 2024
- **Grado**: DGIIM
- **Correo**: quintinmr@correo.ugr.es

Índice

1. Introducción

Dado que un **Archivo Secuencial Físico** se compone de **bloques**, que a su vez se componen de **registros** que se han ido almacenando según un orden secuencial físico, he decidido diseñar **tres TDAs distintos**, cada uno de los cuales representa a una entidad diferente: el archivo secuencial físico, los bloques que constituyen el archivo y los registros dentro de los bloques. Este diseño, a mi juicio, proporciona una mayor modularidad,

así como mayor flexibilidad y escalabilidad, pues podemos agregar nuevas funcionalidades o modificar el comportamiento de cada entidad de forma independiente sin afectar al resto del sistema.

A continuación, se presenta cada uno de los TDA diseñados, junto con su representación en memoria y sus operaciones (cabecera + explicación).

2. Diseño de las estructuras de datos

2.1 Tipo de dato abstracto ASF

Definimos el siguiente tipo de dato abstracto, en lenguaje C++ para la gestión de un fichero ASF:

```
class ASF
{
    private:
        int tam_cabecera;   // tamaño cabecera del fichero
        int num_bloques;    // número de bloques del fichero
        vector<Bloque> bloques; // lista de bloques del
                                fichero
    public:
        ASF();  // constructor sin parámetros
        ASF(int tamanio_cabec, int n_bloques, vector<Bloque>bloqs); // constructor
con parámetros
        void abrir(string rutaFichero); // abrir archivo
        void leerArchivo(string fichero); // leer archivo
        void cerrarArchivo(); // cerrar archivo
        void aniadirBloque(Bloque bloque); // añadir bloque
        void borrarBloque(int identificador_bloque); // borrar bloque
        Registro siguienteRegistro(); // siguiente registro
        Registro obtenerRegistro(float valor_clave); // obtener registro
        void aniadirRegistro(Registro registro); // añadir registro
        void borrarRegistro(int identificador_reg); // borrar registro
        bool buscarValor (int value); // buscar un valor
        float obtenerValor(float value); // obtener un valor
        bool finalArchivo(); // final del archivo
        void modificarCampo (float clave_nueva, float clave_antigua,
                             int identificador_bloque, int identificador_reg, bool
estado); // modificar campo
        void modificarRegistro (Registro nuevo, int identificador_bloque, int
identificador_reg, bool estado);  // actualizar registro


};
```

2.1.1 Representación en memoria

La representación en memoria de un objeto de tipo ASF viene dada por los siguientes parámetros:

- **int tam_cabecera**: tamaño de la cabecera del fichero
- **int num_bloques**: número de bloques del fichero

- **vector<Bloque> bloques**: lista de bloques que conforman el fichero

2.1.2 Operaciones

Presentamos las diversas operaciones que son necesarias para este TDA:

- 1. `void abrir(string rutaFichero);`

  Cabecera: void open(string rutaFichero);

  Especificación: método encargado de abrir el archivo en el sistema de archivos.Establece una conexión entre el programa y el archivo en el disco.

- 2. `void leerArchivo(string fichero);`

  Cabecera: void leerArchivo(string fichero);

  Especificación: método encargado de leer el contenido del archivo desde el discoy traerlo a la memoria del programa para su procesamiento.

- 3. `void cerrarArchivo();`

  Cabecera: void cerrarArchivo();

  Especificación: método encargado de cerrar el archivo.

- 4. `void aniadirBloque(Bloque bloque);`

  Cabecera: void aniadirBloque(Bloque bloque);

  Especificación: Método que añade un bloque (objeto de tipo Bloque) al fichero.

- 5. `void borrarBloque(int identificador_bloque);`

  Cabecera: void borrarBloque(int identificador_bloque);

  Especificación: Método encargado de eliminar un bloque dado su identificador.

- 6. `Registro siguienteRegistro();`

  Cabecera: Registro siguienteRegistro();

  Especificación: devuelve el siguiente registro del fichero respecto de la posición actual del fichero.

- 7. `Registro obtenerRegistro(float valor_clave);`

  Cabecera: Registro obtenerRegistro(float valor_clave);

  Especificación: método que permite recuperar un registro por valor de clave.

- 8. `void aniadirRegistro(Registro registro);`

  Cabecera: void aniadirRegistro(Registro registro);

Especificación: método encargado de ampliar el fichero mediante la inserción de un registro nuevo.

- 9. `void borrarRegistro(int identificador_reg);`

  Cabecera:void borrarRegistro(int identificador_reg);

  Especificación: método encargado de borrar un registro, dado su identificador.

- 10. `void borrarCampo(int identificador_bloque, int identificador_registro, Campo campo_a_borrar);`

  Cabecera: void borrarCampo(int identificador_bloque, int identificador_registro, Campo campo_a_borrar);

  Especificación: método encargado de borrar un campo, dado el registro y el bloque en el que se encuentra.

- 11. `bool buscarValor (float value);`

  Cabecera: bool buscarValor (int value);

  Especificación: busca un valor en el fichero, pasado como argumento. En caso de encontrarlo, devuelve true, y en caso contrario, false.

- 12. `float obtenerValor(float value);`

  Cabecera: float obtenerValor(float value);

  Especificación: método que devuelve el valor especificado como parámetro en caso de que se encuentre en el archivo (llamaría a la función buscarValor).

- 13. `bool finalArchivo();`

  Cabecera: bool finalArchivo();

  Especificación: devuelve true en caso de que estemos en el final del archivo. Devuelve false en caso contrario.

- 14. `void modificarCampo (float clave_nueva, float clave_antigua, int identificador_bloque, int identificador_reg, bool estado);`

  Cabecera: void modificarCampo (float clave_nueva, float clave_antigua, int identificador_bloque, int identificador_reg, bool estado);

  Especificación: método que cambia el valor actual de un campo, en un registro activo, de un bloque, referenciados por los parámetros que se pasan a la función, por un valor nuevo.

- 15. `void modificarRegistro (Registro nuevo, int identificador_bloque, int identificador_reg);`

  Cabecera: void modificarRegistro (Registro nuevo, int identificador_bloque, int identificador_reg);

  Especificación: método que actualiza el registro identificado por identificador_reg, sustituyéndolo por un nuevo registro, en caso de estar en estado activo.

2.2 Tipo de dato abstracto Bloque

Definimos el siguiente tipo de dato abstracto, en lenguaje C++ para la gestión de un fichero ASF:

```
class Bloque
{
    private:
       int identificador_bloque;   // identificador del bloque
       int tam_bloque;         // tamaño del bloque
       int tam_cabec_bloque;   // tamaño cabecera bloque
       int num_registros;   // número de registros del bloque
       vector<Registro> registros; // lista de registros del bloque

    public:
        Bloque(); // constructor sin parámetros
        Bloque(int id, int tam,tam_cabec_bloque, int n_reg,
        vector<Registros> regs);  // constructor con parámetros
};
```

2.2.1 Representación en memoria

La representación en memoria de un objeto de tipo Bloque viene dada por los siguientes parámetros:

- **int tam_bloque**: indica el tamaño del bloque.
- **int tam_cabec_bloque**: indica el tamaño de la cabecera del bloque.
- **int identificador_bloque**: guarda el valor del identificador del bloque.
- **int num_registros**: número de gistros de los que se compone el bloque.
- **vector<Registro> registros**: lista de los registros que componen el bloque.

2.2.2 Operaciones

- `Bloque();`

  Cabecera: Bloque();

  Especificación: constructor sin parámetros del objeto Bloque.

- `Bloque(int id, int tam, int n_reg, vector<Registro> regs);`

  Cabecera: Bloque(int id, int tam, int tam_cabec_bloque, int n_reg, vector<Registro> regs);

  Especificación: constructor con parámetros del objeto Bloque. Un bloque queda definido por su identificador, su tamaño, su tamaño de cabecera, número de registros y los registros que lo conforman.

2.3 Tipo de dato abstracto Registro

Definimos el siguiente tipo de dato abstracto, en lenguaje C++ para la gestión de un registro de un bloque de un fichero ASF:

```
class Registro
{
    private:
      int identificador;   // identificador del registro
      int tam_cabec_registro;   // tamaño de la cabecera del
                                registro
      bool estado;         // estado del registro (true =
                              activo, false = borrado)
      int num_atributos;   // número de atributos del registro
      vector<int> tipo_campo_registro;  // lista con los
                                          tipos de cada campo
                                          del registro:
                                          0 (lógico),
                                          1 (carácter),
                                          2 (entero),
                                          3(real),
                                          4(cadena de
                                          caracteres)
      vector<int> longitud_campo;     // lista con las
                                          longitudes de cada
                                          campo del registro
      vector<Campo> campos;              // valores de los
                                          campos del
                                          registro.

     public:
        Registro();  // constructor sin parámetros
        Registro(int id, int tam_cab, bool state, int n_atrib,
                 vector<int> tipo_campo_reg, vector<int> long_campo,
                 vector<Campo> campos);  // constructor con parámetros

};

// Estructura destinada a la representación de un campo de un
 registro, viendo a un campo como la pareja (attr_id, valor)

struct Campo
{
    int attr_id;  // id_atributo
    float valor;  // valor del campo
};
```

2.3.1 Representación en memoria

La representación en memoria de un objeto de tipo Registro viene dada por los siguientes parámetros:

- **tam_cabec_registro**: indica el tamaño de la cabecera del registro.
- **int num_atributos**: indica el número de atributos propios del registro.
- **vector tipo_campos**: lista de los tipos de los distintos campos del registro: 0(lógico), 1(carácter), 2(entero), 3(real), 4(cadena_caracteres).

- **vector longitud_campo**: contiene las longitudes de los distintos campos que constituyen al registro.
- **int identificador_registro**: guarda el identificador del registro.
- **bool estado**: estado del registro (true=activo, false=borrado).
- **vector<float> campos**: lista en la que se guardan los valores de los campos que constituyen al registro.

2.3.2 Operaciones

Presentamos las diversas operaciones que son necesarias para este TDA:

- ```
  Registro();
  ```

  Cabecera: Registro();

  Especificación: constructor sin parámetros del objeto de tipo Registro.

- ```
  Registro(int id, int tam_cab, bool state, int n_atrib, vector<int> tipo_campo_reg, vector<int> long_campo, vector<Campo> campos);
  ```

  Cabecera: Registro(int id, int tam_cab, bool state, int n_atrib, vector<int> tipo_campo_reg, vector<int> long_campo, vector<Campo> campos);

  Especificación: constructor con parámetros del objeto de tipo Registro.

3. Operaciones que requieren lectura o escritura desde el disco, apertura o cierre del fichero y desplazamiento de la posición actual del fichero.

En esta sección se indican las distintas operaciones de las anteriomente mencionadas que requieren lectura/escritura desde el disco, apertura/cierre del fichero y desplazamiento del offset.

3.1 Operaciones que requieren lectura o escritura desde el disco

Las operaciones que involucran la lectura o escritura de datos desde o hacia el disco son aquellas que acceden al contenido del archivo físico en el sistema de archivos. Es por ello que de entre todas las operaciones, estas son las que requieren lectura/escritura desde el disco:

- La operación **leerArchivo(string fichero)** requiere lectura del contenido del archivo desde el disco y traerlo a la memoria del programa para su procesamiento.
- La operación **cerrarArchivo()** requiere escritura en el disco, pues se necesita escribir cualquier cambio en el archivo.
- Todas las operaciones de inserción y borrado de bloques,registros y campos requieren para ello escritura en el disco, para hacer efectivo el cambio.
- La operación **siguienteRegistro()** para obtener el siguiente registro necesita leer desde el disco, pues en caso de que no se haya leído, y consecuentemente, no esté cargado en memoria, se deberán leer tantos bytes como sea necesario hasta volcar en memoria el registro en cuestión (al menos 12 bytes por campo (4 bytes del attr_id + 8 bytes de valor) más 4 de la cabecera).

- La operación **obtenerRegistro(float valor_clave)** requiere lectura del disco por la misma razón que la operación anterior. En caso de que el registro en el que se encuentra el valor (el registro que estamos buscando), no se haya cargado en memoria, se deberán de leer tantos bytes como sea conveniente hasta que se encuentre el registro en cuestión.

- Las operaciones **buscarValor(float value)** y **obtenerValor(float value)** requieren lectura desde el disco para buscar el valor en el archivo, pues en caso de que no esté el bloque en el que se encuentra dicho valor en memoria, se deberán de leer tantos bytes como sea necesario hasta encontrar el valor.

- Las operaciones **modificarRegistro(Registro nuevo, int identificador_bloque, int identificador_reg)** y **modificarCampo(float clave_nueva, float clave_antigua, int identificador_bloque, int identificador_reg, bool estado)** requieren lectura del disco, en caso de que el registro/campo a modificar no esté cargado en memoria y, por otro lado, requieren escritura, para hacer efectivas las modificaciones.

3.2 Operaciones que requieren apertura o cierre del fichero

Las operaciones que requieren abir/cerrar el fichero para cumplir su función, son las siguientes:

- Las operaciones **abrir(string fichero)** y **cerrar()** requieren apertura y cierre del fichero, respectivamente.

- Con objeto de no ser redundante, como el resto de operaciones en las que se requiere acceso a parte de la memoria del fichero, requieren la apertura del mismo, consideraremos dichas operaciones dentro de este punto.

3.3 Operaciones que requieren desplazamiento de la posición actual del fichero.

Las operaciones que requieren desplazamiento de posición implican mover la posición actual del puntero de lectura/escritura dentro del archivo. De entre las operaciones que he considerado, estas son las que lo requieren:

- La operación **siguienteRegistro()** requiere mover el puntero de lectura hasta encontrar el siguiente registro en el archivo en la dirección de principio a fin. El offset será de tam_registro*numero_registros_visitados bytes.

- La operación **obtenerRegistro(float valor_clave)**. Al buscar un registro por un valor de clave, el puntero de lectura debe desplazarse desde el principio del archivo en dirección al final del mismo para encontrar el registro correspondiente en el archivo. El offset es el mismos que para la operación anterior.

- Las operaciones de inserción requieren mover la posición actual hasta el final del archivo, al ser este un ASF. Se hace en la dirección principio->fin y el offset es tam_registro*num_registros*num_bloques.

- Las operaciones de borrado requiere desplazamiento de la posición actual hacia la posición donde se pretende borrar, siempre y cuando el estado no sea borrado.

- Las operaciones **buscaValor(float value)** y **obtenerValor(float value)** requiere un desplazamiento de la posición actual del fichero en dirección a la posición en la que se encuentra el valor buscado. El offset será tam_registro*num_registros_visitados.

- Las operaciones **modificarRegistro(Registro nuevo, int identificador_bloque, int identificador_reg)** y **modificarCampo(float clave_nueva, float clave_antigua, int identificador_bloque, int identificador_reg, bool estado)** requieren desplazamiento de la posición actual, en la misma dirección que en el apartado anterior, y con el mismo offset.