|  |
| --- |
| עזרא דשט |
| דו"ח לפרוייקט בארגון וניהול קבצים |
| סמסטר ב התשע"א |

|  |
| --- |
| פנחס פדידה 7665692 אבירם מיכאלי 302210224 |

Report in File Management Systems

## Introduction

The goal of this project is to create a system for creating a file that would contain an indefinite number of classes, organized according to the method of hash indexing with linear probing that we learned during the previous semester in File Management Systems and Data Structures: we will insert it in a number according to its key, obtained by a hashing function. We will then be able to find it easily by looking up the place of its key. We are going to give these files the extension ".hash".

The goal of the project is to familiarize us with C++ programming at a higher level than what we used to do in first year and to make us understand the possibilities of this language.

Level 0: Physical File

## Physical File

This class implements our file at the physical level: it enables us to create, close and delete the file, and to read or write plain binary blocks directly into the file. We proceeded step by step by creating each function. There was no need for breaking the problems into sub-problems: the functions were relatively short and each already had a precise task. We did not run into problems testing the functions at debug time either.

### Fields

bool opened;

Boolean field that indicates if the file is already open.

int openmode;

Number that indicates if the file is open for input, output or both.

fstream Filefl;

The file's buffer.

string WorkingDir;

A string that keeps the path of the file it's working on.

string FileName;

A string that keeps the name of the file.

unsigned int FileSize;

Number that represents the size of the file in blocks.

PhysicalBlock FHBuffer;

Buffer for the file header block.

CurrentBlock CurrBlock;

Buffer for the current block.

### API Methods

PhysicalFile(void);

Default constructor, creates an empty physical file class. There is no information on the file yet. The functions popen or pcreate will not immediately be called.

PhysicalFile(string fileName, string workingDir="", int code=2);

Full constructor, creates the physical file and decides whether to actually create the file or to open an existing file using *code* (cf. below). Since the user did not enter *sizeOrMode* as he would have in the next function, a default value is given to it.

Parameters:

* fileName: string, the name of the file.
* workingDir: string, the directory of the file.
* code: integer, 1 if the file doesn't already exists, 2 otherwise. Gives a meaning to the fourth parameter.

PhysicalFile(string fileName, string workingDir, int code, unsigned int sizeOrMode);

Same as above, using a default value for the fourth parameter that indicates sizeOrMode (as said above).

Parameters:

* [Same as above]
* sizeOrMode: positive integer, indicates the size of the file in blocks if the value of code is 0 and the opening mode (read/write/both) if it is 1.

~PhysicalFile(void);

Destructor, destroys the physical file class by closing the file if it still open (using *opened*) and still exists (using *fileName* to check if the name is still assigned, which will be undone if the user calls pdelete, destroying the file).

void pcreate(string fileName, unsigned int fileSize=1000, string workingDir="");

Should only be called once for each file. Creates the file and assigns the parameters to their equivalent variables in the class. The blocks are subsequently all initialized to zeros.

Parameters:

* fileName: string, the name of the file.
* fileSize: positive integer, the size of the file in blocks.
* workingDir: string, the directory of the file.

void pdelete(void);

Deletes the actual file: the data is lost.

void popen(string fileName, int openMode=0, string workingDir="");

Opens the file through fstream.

void pclose(void);

Closes the file if it is still open (using *opened*).

void writeBlock(unsigned int position);

Writes a physical block at the desired position.

Parameter:

* position: positive integer, indicates the serial number of the block.

void readBlock(unsigned int position);

Writes a physical block at the buffer's present position.

Parameter:

* [Same as above]

void writeBlock(void);

Writes a physical block at the buffer's present position.

void readBlock(void);

Writes a physical block at the desired position.

void writeFH(void);

Writes the file header through *FHBuffer* at position 0.

void readFH(void);

Reads the file header to *FHBuffer*.

### Helping functions

void SeekToBlock(unsigned int position);

Moves the file buffer and the current block buffer using seekp and seekg to the desired position.

Parameter:

* position: positive integer, indicates the serial number of the block.

bool FileExists(string filePath);

Checks if the file already exists, to prevent wrongful openings.

Parameter:

* filePath: string, indicates the full path of the file (name and directory).

void InitializeData(void);

Used by pcreate, initilalizes all the data blocks to strings of zeros.

Level 1: Logical File

## Logical File

This class implements our file at the logical level: it is responsible for filling and manipulating the file header and the logical blocks through the physical level. Its main task is to maintain the buffers and flushing them before closing the file.

### Fields

string UserName;

A string that keeps the name of the user that created the file in memory, for instance to check if the file belongs to the user trying to write into it.

LogicalBlock\* LogicalBuffer;

A buffer containing the present logical block.

LogicalFileHeader\* LogicalFHBuffer;

A buffer containing the file header for reading, writing or updating the file information.

bool updateflag;

Boolean field that indicates whether or not the last block that was read has been updated.

bool LBuffChanged;

Indicates if the logical buffer was changed, thus needing to be written at flushing.

bool LHBuffChanged;

Same as above, for the logical header buffer: indicates if the file header has been updated since last time it was read.

unsigned int CurrRecNrInBuffer;

Positive integer that indicates the serial number of the record that is currently in the buffer from the beginning of the block.

### API Methods

HashFile(void);

Default constructor, creates an empty logical file class. As for PhysicalFile's default constructor, there is no information on any file yet. However, some variables receive their default value.

HashFile(string fileName, string userName, unsigned int recordSize, string workingDir="", int code=2);

Partial constructor that does not receive any parameter following *code*. It gives every other variable that is present in the following function its default value. Refer below.

Parameters:

* fileName: string, the name of the file.
* userName:string, the name of the user.
* recordSize: positive integer, the size of each record in bytes.
* workingDir: string, the directory where the file is to be created / is currently present.
* code: integer, gives a meaning to the *sizeOrMode* parameter, just as in PhysicalFile's partial constructor.

HashFile(string fileName, string userName, unsigned int recordSize, string workingDir, int code, unsigned int sizeOrMode, int keyPlace=0, char keyType='I', int keySize=4);

Full constructor: it receives enough parameters to link the class to a file. It also fills its file header and initializes the Boolean fields, subsequently opening or creating the file on the logical file.

Parameters:

* fileName: string, the name of the file.
* userName:string, the name of the user.
* recordSize: positive integer, the size of each record in bytes.
* workingDir: string, the directory where the file is to be created / is currently present.
* code: integer, gives a meaning to the *sizeOrMode* parameter, just as in PhysicalFile's constructor.
* sizeOrMode: positive integer, represents the size of the file in blocks or the opening mode, depending on the value of *code*.
* keyPlace: integer, the position of the hash key inside the record.
* keyType: character, represents the type of the hash key: 'I' for integer, 'S' for string.
* keySize: integer, the size of the key in bytes.

~HashFile(void);

Closes the file, preparing for the destroying of the object.

void hcreate(string fileName, string userName, unsigned int recordSize, string WorkingDir="", unsigned int fileSize=1000, unsigned int keyPlace=0, char keyType='I', unsigned int keySize=4);

Creates the file on the logical level (calling *pcreate* on the physical level using its relevant parameters) and fills its file header using the *LogicalFHBuff* buffer and *writeFH*.

Parameters:

* fileName: string, the name of the file.
* userName:string, the name of the user.
* recordSize: positive integer, the size of each record in bytes.
* workingDir: string, the directory where the file is to be created / is currently present.
* fileSize: integer, the size of the file in blocks.
* keyPlace: integer, the position of the hash key inside the record.
* keyType: character, represents the type of the hash key: 'I' for integer, 'S' for string.
* keySize: integer, the size of the key in bytes.

void hdelete(void);

Calls *pdelete* in order to destroy the file on the physical level.

void hopen(string fileName, string userName, string workingDir, int openMode);

Opens the file on the logical level through *popen* and checks whether or not the user has the required authorizations to write into it if he so wishes using the provided *userName* and the *UserName* field recuperated from the file header.

Parameters:

* fileName: string, the name of the file.
* userName:string, the name of the user.
* workingDir: string, the directory where the file is to be created / is currently present.
* openMode: integer, indicates whether the file is to be open in input/output/both.

void hclose(void);

Flushes the buffers and then calls *pclose* in order to close the file on the physical level.

void flush(int buffer=1);

Writes the blocks in the buffer that have been updated since they were read and not yet written into the file.

Parameter:

* buffer: integer, indicates which buffer/s is/are to be flushed.

## Physical Block

We began by implementing the physical blocks as a structure, which is consistent with their essence of being a tool and not a "machine": therefore, they do not encapsulation. It is however important that we keep the same size for all the types of blocks that we will define in the project. We chose a size of 1024 bytes: one Kilobyte (Kibibyte according to the SI).

### Fields

unsigned int BlockNr;

Positive integer, the serial number of the block.

char Filler[20];

String of bytes, represents filler bytes.

char Data[1000];

String of bytes, the future data in the blocks of the following layers.

## Logical Block

### Fields

unsigned int BlockNr;

Positive integer, the serial number of the block.

unsigned int NrOfOverflowedRecs;

Positive integer, the number of overflowed records (records that couldn't fit in their block).

unsigned char NrOfRecsInBlock;

Positive integer, the number of records currently in the block..

char Filler[10];

String of bytes, represents filler bytes.

char Data[1000];

String of bytes, the future data in the blocks of the following layers.

Level 2: Manipulating the records

## Seek, Write and Read

For this level, we will stick to the same classes (PhysicalFile, HashFIle and the different Block structures) and we will add a few functions that allow us to manipulate the records. Note that all three functions have three implementations: with C++ string, C string and integer keys. In order to write a record, we check if another with the same key hasn’t already been written in the file, and then proceed to write the record in the relevant block. And in order to read one, we call seek to get its position and then return a pointer to the record to the main.

### API Methods

bool seek(string& key);

bool seek(char\* key);

bool seek(int key);

Checks if the record with the given key is present in the file. It retrieves the number of the block by using the HashValue class contained within the HashFile class and then checks if the record with the relevant key is present within the said block. If it failed to find the record and the block may have overflowed, then it does the same thing for all the blocks in the file [once]. The Boolean is then returned: its value is true if the record has been found and false otherwise.

Parameter:

* key: C++ string / C string / integer, depending on the key type. This is the unique key of the record to be found.

void write(string& key, char\* record);

void write(char\* key, char\* record);

void write(int key, char\* record);

This function receives the key of the record and a pointer to the record itself in bytes. It then calls *seek* in order to check if a record with the same hasn’t already been written in the file. If seek returns true, then such a record indeed exists and an exception is thrown. Otherwise, it looks for a proper place to write the record iteratively: the first block that is not full, from the one pointed to by *HashValue*. The record is then written in a proper empty place.

Parameters:

* key: C++ string / C string / integer, depending on the key type. This is the unique key of the record to be written.
* record: a pointer to the record in byte shape. Handy for a memcpy to the proper place in the *Block*’s *Data* field.

void read(string& key, char\* record, int readType=0);

void read(char\* key, char\* record, int readType=0);

void read(int key, char\* record, int readType=0);

This function very simply calls *seek* in order to check if the record is in the file and to place the buffer at the proper position at the same time. Afterwards, the record is copied in a char pointer and sent to the caller.

Parameters:

* key: C++ string / C string / integer, depending on the key type. This is the unique key of the record to be written.
* record: a pointer to the record in byte shape. Handy for a memcpy to the proper place in the Block’s Data field.
* readType: integer, indicates if the record will be read only (0, value by default) or also updated (1). In the latter case, the user's authorization to write into the file are to be checked, as it was done in *hopen*.