**מר עזרא דשט - ד"ר משה גולדשטיין**

**דו"ח מסכם של המיני פרוייקט בארגון וניהול קבצים**

**סמסטר ב התשע"א**

**פנחס פדידה 7665692 אבירם מיכאלי 302210224**

Table of Contents

**Introduction3**

**Level 0: Physical File3**

**Level 1: Logical File6**

**Level 2: Manipulating the record9**

**Level 3: Deleting and updating records11**

**Level 4: Graphical User Interface12**

**How to use the Final GUI version13**

**Code ameliorations**15

**Recommendations for another level**16

**Thought on the course**17

**Thoughts on the teachers**17

**Conclusion**17

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*.

Level 3: Deleting and updating records

## delrec, update, updateoff

For this level, we will still stick to the same classes (*PhysicalFile*, *HashFIle* and the different *Block* structures) and add 3 functions to the *HashFile* class that will together allow us to "physically" delete and to update any record that has already been written into the file, or at least into the buffer: *delrec* deletes a record by overwriting the bits with last record's, *update* updates a record by replacing the bits with a given. Both functions use the *updateflag* parameter in the *HashFile* class in order to lock the record while reading the record using the proper parameter in read (that was written in Level 2). This ensures that following this reading, no parameters will change until the record is unlocked, using the last of the three functions: *updateoff*.

### API Methods

void delrec(void);

Deletes the record that is present in the buffer "physically" (bit-per-bit overwriting) instead of "logically" (marking the bits as invalid): the function read has to be called for update previously (using the third parameter, cf. Level 2), thereby locking the buffer on the record that has just been read. It then proceeds to read the record's key for future use ([see further](#Key)) and overwrites the record's bits with the last record in the block using *memcpy*. The block's and the file header's variables are then updated (*NrOfRecsInBlock* and *LBuffChanged*, *NrOfRecsInFile* and *LHBuffChanged*, respectively). If the record should have originally been in another block (using the key that we previously read), this block is read in the buffer (using *HashFunction* to get its position), and its *NrOfOverflowedRecs* and *LHBuffChanged* variables are then updated accordingly.

void update(char\* record);

This function updates a record by overwriting the old record's bytes with the new ones: as in delrec, the function read has to be called for update previously (using the third parameter, cf. Level 2), thereby locking the buffer on the record that has just been read. Using the pointer to the updated record received as a parameter, it then checks if its key is identical to the new record and continues (attempting to update a record with another record that has a different key is an illegal action). As everything is ready, the old record's bits are overwritten by the updated one's using *memcpy*. The block's LHBuffChanged is then updated to true, as is usual when the buffer is modified and not immediately written.

Parameters:

* record: a pointer to the record that contains the new data in byte shape.

void updateoff(void);

This function unlocks the class that was previously locked for update by turning the updateflag class inner variables from true to false.

Level 4 : Graphical User interface

This final level will take us to the graphical layer: we are going to present the application to the user in a window, which makes it more intuitive to navigate between files and methods, and gives a solid feeling of encapsulation through the graphic components. This level has been coded in C++ and Managed C++ using a Windows Forms application with the .NET common language runtime (CLR) environment.

Now let us give some detail on the approach of this final level: At first, I tried to create a Microsoft Office approach where changes were stored in a temporary file until they were saved by the user. Due to a lack of time, we switched to a more classic approach: changes are executed immediately on the file itself upon calling one of Level 0’s writeBlock functions. Apart from that, the user can create, open or delete a file. Record actions (seek, write, update, delete) are made on the main form instead of navigating in endless dropdown menus. Finally, the user is exceptionally allowed to have a peek at the file structure thanks to the ListView controls that represent a file map, and allow the user to know how many and which records are in any given block in the file that is currently opened. Other minor useful controls like empty volume are present in a file info group box and show valuable information about the currently opened file.

I do not judge it necessary to detail the various fonctions used in the GUI, as they have absolutely no algorithmic interest and are only relevant to the programmer interested in Managed C++.

How to use the final GUI version

About

File

How to use

About

Hash Doc

Open

Create

Delete

Exit

Key

First name

Key

City

**Delete**

**Update**

Last name

Last name

City

First name

Key

**Write**

Layer 0-1

**Read**

Layer 2

File info: name, size, key type, etc.

Layer 3

**How much Empty**

Layer 4 - GUI

Record list

58 Avi Cohen Ashdod

45 David Peres Tel Aviv

Block list

1. ■■ .
3. ■■■■■■■

**File map**

## The menu strip

* File: Logical and physical actions on the .hash files can be found in this submenu: you can create, open, or delete opened files in a simple manner. You can also find the Exit button, as is usual in Windows systems.
* About: Short but instant information about the application can be found in this submenu: the how-to-use dialog, similar to this one, the Hash Documentation (from the introduction to HashValue.h, used for the mini-project), and the About dialog, that delivers usual information about the application: aim, version, disclaimer and copyright.

## The main form

* Upper controls: On the upper part of the main form are a few upper controls that allow the user to execute basic actions on the records, as described in layer 2 and 3: read, write, update and delete, depending on the open mode the user chose when he opened the .hash file.

A word on the records: using the source files, we can of course change the record type that the we can store within the .hash file, but this application's GUI was especially coded for one type of 128-byte record that was used for testing purposes in the lower layers, representing an individual's location card and containing 4 fields: a unique key, a first name, a last name and a city.

There is a group box with useful information about the file opened. One interesting feature is that it displays the percentage of place available.

* Lower controls: One of the newer features in the final version is the file map: from there, the user can easily see how the .hash file he is editing is structured: once the file is opened, the block list will fill with the 1024-byte block's number and a number of squares representing in a visual manner how many records are in the said block (cf. on the left of the schema). If the user clicks on a block in the block list, he will see the contents (if there are any) of this block in the record list on the right, with the fields of the records present in the file, as we described earlier. Also, a handy feature is that if the user click on a record in the record list, its key will appear in the 'Seek' key field, for easier editing.

Note: sometimes, when the application is required to do a lot of actions, a bad allocation exception pops up, in innocent allocations where it is obvious that there is no mistake, as if the computer suddenly ran out of memory. And this kept happening in level 3 and level 4. *But once it happens, every following allocation will be a bad allocation*! As if the program was ordered to stop allocating after a defined amount of memory. If you also stumble upon this and can fix this, please contact me at fedidat@gmail.com.

Code ameliorations

Here are some changes that were made in our application compared to the original instructions. No offense meant, of course, but some of these made sense, while others were algorithmically efficient:

* Let us begin with one major decision: HashFile inherits PhysicalFile, instead of the former containing a pointer to the latter. Shortly, although this makes seem counter-intuitive at first, because it's a different system, it actually makes a lot of sense: HashFile uses PhysicalFile on another level, it is a more 'advanced' implementation of PhysicalFile, and as such, HashFile Is the logical extension of PhysicalFile. Thus, it is consistant that HashFile inherits PhysicalFile, because this is basically the meaning of inheritance.
* Now, from the beginning: I turned a number of variables into pointers (blocks, for instance), because they are essentially a pointer to the memory slot they hold. This also avoids cases of concurrence between variables that overlap (notably in buffers). The same goes for the HashValue component of HashFile, *hashVal* (cf. Level 2), because it needs to be initialized in order to use the HashFunction feature (otherwise, the lowest prime number is not initialized).
* One important change that is crucial in terms of CPU times is the decision to get rid of the endless calls on the flush function, that are also not present enough and to call flush in HashFile's readBlock. This is an excellent decision in my opinion because it is very consistent with the way actual paging works in operating systems. And it allows us to just turn on a buffer's flag to be absolutely sure that it will be updated, without taking any risk on continuity.
* Many times, it was suggested in the guide to make useless calls to function. And several times, we were told to turn on buffer flags for nothing. In fact, if you look at my implementation of Level 2, you will see a good number of differences with the instructions' suggestions, in the good way.
* Some functions, such as seek, delete and update, had to be greatly changed in order to make them functional: reading and writing the key from the pointer to record is in itself a challenge, and the binary manipulation is generally different from what is described in the guide.
* Instead of seek throwing an exception when the record is not found, I made it return a bool. That way, we do not have to catch the exception when we write a record (in which case it is essential that there is not another record with the same key). This makes sense, because in that case, it is not an error, but it works in a similar manner than the way 'find()' works in several C++ STL components.
* A significant number of mistakes in the PDF guide to mini-project. Most of them were obvious, but some turned out to be a challenge.

Recommendations for another level

When my USB pen was stolen, I attempted a completely different approach of my GUI: I tried to make it work like modern editing software. For understanding purposes, let us consider Microsoft Office:

* When the user creates a file, he needs not provide any details except for the type of the file. But the file name and the folder will only be provided when the user saves his file, either by using save or save as, which allows to make a copy of the file in buffer as a totally different file.
* In order to accomplish this, Office uses temporary files: when the user opens of creates a file, the program creates an invisible temporary file, which will only be removed when the program is closed.
* This allowed another, much greater change: the user can undo any changes made to a file, as they are stored in a buffer, and will only be applied when the user saves his file (and occasionally, during autosaves).
* Unfortunately, the lower layers were not designed for this and I stumbled across obstacles (alas I do not remember which ones) and had to write my program for the third time in order to obtain the result that you saw.

Thus, this is my recommendation: get the students to make a modern-looking application by adding the possibility to undo changes. I realize that this is a different approach of the program, but this is how modern software works.

As this is not really a recommendation for a level, an alternative idea would be to make a middle layer that can compute things between records through a sort of SQL language, in order to finalize the tidy database aspect of the mini-project.

Thoughts on the course

In my opinion, this course was very instructive and gave an excellent insight into the world of programming: this mini-project truly looks like a small example of what students will be required to do as a programmer. It took an incredible amount of time to figure out what we were meant to do in level 2, and it is a good example of algorithmic thinking: how we can break things down into a multitude of smaller tasks and make it much simpler. That was a long project but the lessons are learnt.

Thoughts on the teachers

Mr. Ezra Dasht guided through the first half of the semester. We could feel the programmer's approach, simplifying every problem: to every question there was a simple and obvious answer.

But Ezra fell sick and Dr. Moshe Goldstein replaced him. There came the doctor's approach, more methodic. The answer was not obvious anymore. There were steps to be followed. But there was always a solution, and it is in somewhere in the programmer's code.

I would like to thank both teachers for teaching us their approach and helping us through the mini-project.

Conclusion

Level after level, we went from the physical layer, where we could feel the binary, to the logical layer, where we manipulated chunks of data, to the graphical interface, where the encapsulation finally hid everything in the box. And we were able to contemplate the steps through which every piece of software goes, from zeros and ones to windows.