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

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

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*.

## Helping functions

bool seekInBlock(int position, char\* key);

bool seekInBlock(int position, int key);

This function allows us to write a simpler *seek* function, by calling this specific function for every block where we will be looking for a record.

Parameters:

* position: integer, indicates the number of the data block where we are to seek the record with the given key.
* key: C string / integer, depending on the key type. This is the unique key of the record to be written.

bool writeInBlock(int position, char\* record);

This function allows us to write simpler write functions, by calling this function every time we want to write in a specific block.

Parameters:

* position: integer, the number of the data block where we are to write the record.
* record: a pointer to the record in bytes.

## Testing the Hash functions

Some testing with the functions written in the MainProgram.cpp of this level can reveal that, as the author of the *HashValue* class stated in his introduction to *HashValue.h*, there is no "perfect hash function". In fact, when using random keys, the results can vary from run to run.

After some tweaking, I created a function that would run each hash test function 100 times. Some functions seldom get the first place (such as Weinberger's), but in general, the result for both time and deviation depends much more on how hard the computer is working that on which function you choose. There is no significant benefit unless you are going to enter your records according to a pattern that tends to identify to some hash function.