Ονοματεπώνυμα και ΑΜ
Αντώνιος Καρβελάς 1115201600060
Παντελεήμων Μαλέκας 1115201600268
Θεοφάνης Μπιρμπίλης 1115201600110

Here's all the information regarding what was implemented in the hash_file.c:

Regarding the array of the open files we used a global array, named tableOfIndexes. Each item in this array consists of a struct named openedIndex, which contains the file descriptor and the number of buckets of each file.

1) HT_Init()

In this function, the array of the open files is initialized, setting each entry with the NULL value.

2) HT_CreateIndex(const char* filename, int buckets)

In this function the hash file is created, and the metadata is placed inside the file. First, a "HT" string is placed to signify that it's a hash file and after that we place the number of buckets for this file. Also, in this function we allocate the blocks needed for the Index. Since every entry in the index will have an integer (in this implementation we place the block number of the first allocated block in the bucket), we calculate how many integers can be placed inside the block and we allocate the required amount of blocks for the index.

3) HT_OpenIndex(const char* filename, int *indexDesc)

This function opens the hash file and checks if we have a hash file and whether it can be opened or not. If so, an openedIndex struct is allocated and it's placed in the array of open files.

4) HT_CloseFile(int indexDesc)

This function finds the file in the array of open files and closes the hash file using its file descriptor.

5) HT_InsertEntry(int indexDesc, Record record)

This function gets the hash file from the tableOfIndexes array. We get the hash value of the record using modulo operation on the number of buckets. Then, we calculate which block from the index we will have to go to and place the record. After getting the record's position in the block we check the bucket's data. Here, we have two cases: a) if the bucket data is equal to 0 this means no record has been placed in this bucket, so we allocate the first block and place the record. Each block also has a counter with the number of records inside it as well as the block number of the next block in the bucket. Default value of this variable is -1. b) if the bucket data is not 0, this means we have at least one record in this bucket. So here, we get the last block in the bucket and how many records are in this block. If it's full, we allocate a new block to place the record and we place its block number in the previous block. If it's not full, the record is placed in the first available slot.

6) HT_PrintAllEntries(int indexDesc, int *id)

Similar logic as in InsertEntry is used here in order to find where the record is placed in the hash file. If the id's value is NULL the function will print every record from the hash file. If not, we get which bucket contains the record and search along its blocks to find the record and print it. If we reached the final block and we didn't find the record this means this record doesn't exist in the hash file, so an appropriate message is printed.

7) HT_DeleteEntry(int indexDesc, int id)

In this function we find the desired record inside the hash file, just as in the two previous functions. If the record isn't inside the hash file an appropriate message is printed and we exit the function. If we found it, we get the last record from the same bucket. In order to delete the record, we write the last record on top of the record to delete and we decrease the last block's counter.

Three main functions were created in order to validate the accuracy of the program.

The first main function (ht_main.c) is almost the same as the given one, only here we call the PrintAllEntries and DeleteEntry functions a few more times, with different arguments. To run this main function, type: ./build/runner

The second main function is main_one_file.c. Here we place a large number of records in one hash file and perform a large number of deletions. Also, the number of buckets is larger than the default one(which is 13), to showcase multiple blocks being created for the index. To run this main function, type: ./build/runner2

In the third main function, main_multiple_files.c, we open the maximum number of files (Since this number is set to 20 in the hash_file.c, if one wishes to change it, be sure to also check the number of files being opened in the main_multiple_files.c and change it if need be). Since MAX_OPEN_FILES can be opened at a time, the main function won't open any more files if we exceed the limit. This function also closes the first file in the first iteration, showing that it can be opened again later. Finally, each hash file has its own number of buckets. This number is set to 13 for the first file and is incremented by 50 for each consecutive file. To run this main function, type: ./build/runner3