Assignment 2: Heap File Implementation

Due Friday, July 23th

The purpose of this relatively straightforward assignment is to begin the implementation of the class `DBFile`, which will encapsulate the management of a single file in your database system. If someone using this class wants to open up a bunch of database files, they will need several instances of the `DBFile` class in order to do this. The class `DBFile` will run on top of your buffer manager, so any operations to files should go through your buffer manager, and they should not be written directly to disk. The header for the `DBFile` class is as follows:

```
extern LRU myBuffer; // this is the buffer manager

enum FileType {

 Sorted,

 Heap,

 BTree

 //Note, we will use Sorted and BTree types in the next assignment.

};


Class DBFile {

private:
 // whatever you need in here!

public:
        int Create (char *fName, Schema &schema, int
                        pagesToAllocate, FileType fType);
        int Open (char *fName, Schema &schema, int fileSize,
                        int firstPos, FileType fType,
                        int *numDistincts, int numTuples);
         int Find (Value &key);
         int GetNext(Record &fillMe);
         void Reset ()
         void Insert (Record &insertMe);
         void GetInfo(char * &fName, Schema &schema,
                        int &fileSize, int &firstPos,
                        FileType &myType, int *&numDistincts,
                        int &numTuples);
         void Close ();
         DBFile ();
         ~DBFile ();
};
```

The Specification

For this particular assignment, you only need to consider the heap file type. Implementing the `Sorted` and `BTree` are saved for the next assignment. Also note that in this assignment, **you should not ever modify the catalog.** There are ten methods for the `DBFile` class that need to be implemented. They are:

`int Create (char *fName, Schema &schema, int pagesToAllocate, FileType fType)`

This function creates a new database file with the type and schema specified by going to the buffer manager and setting up the file (in this assignment the type is always `Heap`). The buffer manager should allocate `pagesToAllocate` pages for the file (this is the `numPages` argument to the buffer manager), and the original size of the file is zero. This routine should return a 1 on success and a 0 on failure (for example, if the path specified for the file is not available).

`int Open (char *fName, Schema &schema, int fileSize, int firstPos, FileType fType, int *numDistincts, int numTuples)`

This function simply opens the specified file. The user gives you the path for the file. The user also gives you the schema for the file, the size of the file at the time that it was closed (in terms of the number of pages), the "first position" in the file (which will always be 0 for a heap), and the type of the file.

There are two additional parameters that you will need to deal with. The second is `numTuples`, which simply tells you the number of records in the file (you will need to update this value internally inside of the class as records are added). The first is `numDistincts`, which is an integer array that tells you the number of distinct values for each of this file's attributes. For example, if `numDistincts` is [2, 4, 3] this means that there are 2 distinct values for the first attribute, 4 for the second, and 3 for the third. You won't actually use these values or alter them in this assignment (they will be used later on for query optimization), and you don't have to update them as new values are inserted, but you do need to remember them and then give them back to the user when asked (see the `GetInfo` routine below).

This routine should return a 1 on success and a 0 on failure (for example, if the file specified does not already exist).

`int Find (Value &key);`

The `DBFile` class maintains a "pointer" to the current record in the file (kind of like the buffer manager maintains a "pointer" to the current page in the file). The pointer is set to the first record in the file when it is opened. This particular function moves the pointer to point at the first record in the file that has the specified attribute value, and returns a 1 if it finds such a record. If there is no such record, it points to the first record in the file that has a key value greater than the specified attribute value. In this case, the function returns a 0. If there is no record in the file that has a larger value, then the pointer points just past the last record in the file. In this case, this function returns a –1. For heap file you will simply scan the pages in the file from first to last, one by one, in order to find the record in question.

Note that an object of type `Value` has three important constituent properties: an actual value (for example `3`, or `2.45`, or "`this is a value`"), a type (`Int`, `Real`, or `CharString`), and an attribute name. When your `DBFile` is asked to find a given value in the file, you need to make sure to search on the correct attribute (the one that matches the attribute name of the parameter `key`). If someone asks you to search on an attribute that does not exist, or on an attribute that causes a type mismatch, then `Find` should also return a –1.

`int GetNext (Record &fillMe);`

This function copies the record that is currently under the pointer into the parameter `fillMe`. The pointer is then moved to the next (physically adjacent) record in the file. The function returns a 1 on success. It returns a 0 if the pointer points past the end of the file, when the function was called (so that it cannot return another record).

`void Reset ();`

This function moves the pointer to point at the first record in the file.

`void Insert (Record &insertMe);`

This function adds the given record to the file. For heap file type, the record simply is appended to the end of the last page in the file.

`void GetInfo (char * &fName, Schema &schema, int &fileSize, int &firstPos, FileType &myType, int *&numDistincts, int &numTuples);`

This function produces values for these parameters, according to the current state of the file. Note that `GetInfo` will need to allocate memory to set up the return the values `fName` and `numDistincts`, since the caller only sends in references to pointers to these parameters.

`Void Close ();`

`DBFile ();`

`~DBFile ();`

These functions are straightforward. `Close` should get the LRU buffer manager to close the file. `DBFile` should close the file if `Close` has not been called previously.

Appendix

This assignment comes along with some simple classes implemented to parse the catalog specification, build schema etc. The `Catalog` class will be used to read/write information about a file to/from the system catalog. This class essentially allows all of the info about a `DBFile` object to be written to a text file so that it can be saved for the next time that the database is started up. The file format of catalog specification (as it appears in the catalog) is given below. You can easily create your own catalog files in accordance with this format. However, since we have already

implemented the `Catalog` class, you will not actually have to read this stuff in… you just ask the `Catalog` class to do it.

`BEGIN FILE`

`fName` <name of the file>

`relationName` <name of the relation stored in the file>

`5` <number of attributes in the relation>

`INTEGER` attName <type of the first att, and its name>

`REAL` attName <type of the second att, and its name>

`STRING 30` attName <and so on, for all of the attributes>

`INTEGER` attName

`REAL` attName

`KEY` attName <tells what the "key" of the relation is>

100 <stat data: last know number of distinct values for att 1 in the file >

200 <stat data: last know number of distinct values for att 2 in the file >

335 <stat data: last know number of distinct values for att 3 in the file >

300 <stat data: last know number of distinct values for att 4 in the file >

400 <stat data: last know number of distinct values for att 5 in the file >

10000 <stat data: last known number of tuples in the file>

`2655` <number of **used** pages/blocks in the file>

`0 <starting block of the file>`

`fType` <this is the type of the file: for this assignment, either `SORTED` or `HEAP`>

`END FILE`