University of Edinburgh
School of Informatics
INFR11011 - Advanced Databases (Spring 2020)

Coursework Assignment
Due: **Thursday, 26 March 2020 at 4:00pm**

IMPORTANT:

- **Plagiarism:** Every student has to work **individually** on this assignment.

  All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. You may not host your code on a public code repository. Plagiarism **will not** be tolerated.

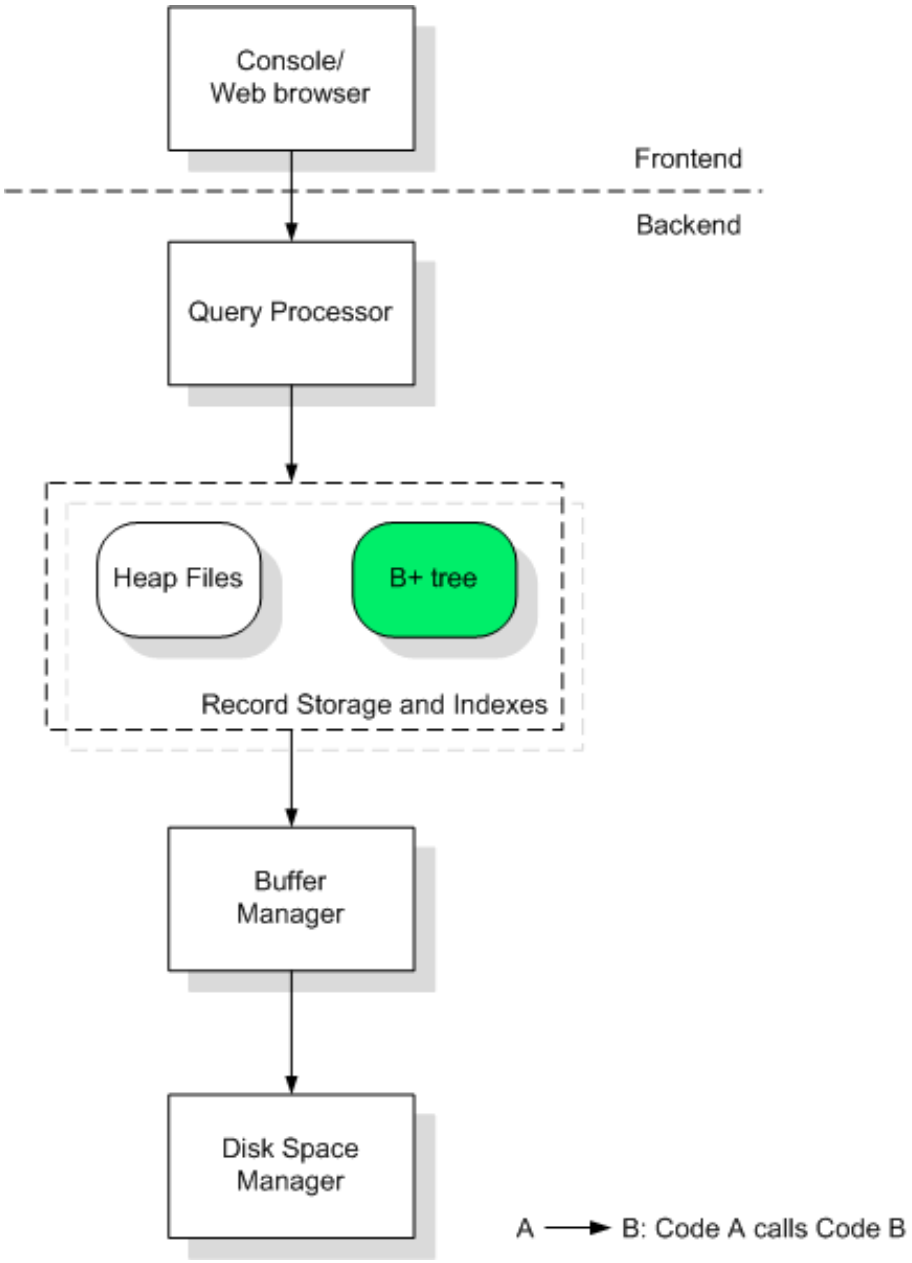
  See the University's page on Academic Misconduct for additional information.

- Start early and proceed in steps. Read the assignment description and the FAQ carefully before you start programming.

- This assignment is worth 30% of your overall grade.

Project: B+ Tree

Your goal is to implement a B+-tree index structure in the MINIBASE database system.



A B+-tree index consists of a collection of records of the form `(key, rid)`, where `key` is a value for the search key of the index, and `rid` is the record id of the record being indexed. (It is an index based on what is described as Alternative 2 in the textbook.)

The nodes in a B+-tree can be divided into two categories: internal index nodes and leaf nodes. The leaf nodes store the actual `(key, rid)` pairs, while the internal nodes store key values and the page identifiers (page ids) of their children.

Read the textbook carefully and familiarize yourself with the B+-tree indexing structure and algorithm.

Overview of MINIBASE Components

We start with an overview of components inside the MINIBASE database system. MINIBASE provides the Database (DB), Page, and Buffer Manager abstractions. The formative assignment on the heap page implementation already used these abstractions.

As you may recall, the DB abstraction is responsible for maintaining file entries, allocation/deallocation of pages, and performing I/O on the pages. The Page abstraction is just a dummy class of size `MAX_SPACE`, and `HeapPage` is a special type of Page. The Buffer Manager is responsible for maintaining "frequently" used pages in memory. In this assignment you will have to build a B+ tree on top of these abstractions.

Space Manager

The space manager is responsible for keeping records in a page. You may already be familiar with this component since the formative assignment was about implementing the heap page format with a slot directory. In this assignment, the implementation of `HeapPage` is given to you.

- A *heap page* maintains records using a slot directory located at the beginning of the page. Each slot contains two attributes: `offset` and `length`. Records starts from the end of the page. Offset refers to the offset of the record within the page, and length refers to the length of the record. See `include/heappage.h`.
- Each page is uniquely identified by a page id. A record id is a pair (page id, slot number) that uniquely identifies a record.

  The identifier of a page, the page id, is implemented as type `PageID`, which is just typedef as `int`. The identifier of a record, the record id, is implemented as a structure `RecordID`, containing two fields: `pageNo` of type `PageID` and `slotNo` of type `int`. Comparison operators == and != are also implemented on `RecordID`. See `include/minirel.h`.
- The class `HeapPage` implements a heap page and provides interface to insert and delete records from the page, via `InsertRecord()` and `DeleteRecord()`. It also provides functions to scan the page, via `FirstRecord()` and `NextRecord()`.

  The class `HeapPage` also has a member called `type`. In this assignment, the type of a page indicates whether the page is a leaf page or an index page of the B+ tree.
- `SortedPage` is a special type of `HeapPage` that maintains records in a page in increasing sorted order based on a key. Insertion sort is used to insert the records, so the records in a `HeapPage` may be moved whenever insertion occurs. This may change the `RecordID`s of the records. For this reason, the `RecordID`s of records in a `SortedPage` should never be exposed to users.

  See `include/sortedpage.h` and `src/sortedpage.cpp` for more details.

Buffer Manager

The buffer manager is responsible for bringing pages from the disk to the main memory as needed. As you may recall from the lectures, the buffer manager:

- allocates a buffer pool and brings the requested pages from the disk to the pool;
- pins a page to the buffer pool, bringing a page from the disk, if necessary;
- unpins a page in the buffer pool by reducing the pin counter for that page;
- frees up space that is allocated to a given page on the disk;
- flushes a page to the disk, if required;
- keeps track of statistics of operations on the buffer pool.

You will often use pin and unpin operations in this assignment. Use `PinPage()` when you have a `PageID`, and you want to read in the page from disk into memory. After you use the page, you must remember to `UnpinPage()` with the correct dirty flag.

The buffer manager will also create a new page via `NewPage()` whenever you need one (e.g., when you create a B+ tree). Remember to free the page when you do not need it anymore (e.g., when you delete a page from the B+ tree), by calling `FreePage()`.

See `include/bufmgr.h` for the buffer manager API.

Overview of the B+ Tree

A B+ tree index is stored in MINIBASE as a database file. The class `IndexFile` is an abstract class for indices. `BTreeFile` is a subclass of `IndexFile`, and implements the B+-tree in MINIBASE. `BTreeFile` maintains two types of nodes: `BTLeafPage` and `BTIndexPage`, which correspond to the leaf nodes and index (internal) nodes in a B+-tree.

The `BTreeFile` class provides methods for inserting, deleting, and searching the tree. It does not contain the tree itself. It maintains the tree by keeping a pointer to the root of the tree. The `BTreeFile` class is responsible for maintaining the integrity of the tree (i.e., when an insertion is made, it has to ensure that the new record is inserted in the correct place; or if the current node does not have enough space to hold the record, it should perform node split).

The leaf and index nodes in a B+ tree are implemented by the classes `BTLeafPage` and `BTIndexPage`, respectively. They are both subclasses of `SortedPage` and are responsible for inserting into and deleting from leaf node and index node, respectively. These two classes are not responsible for the integrity of the tree.

For this assignment, `BTLeafPage` and `BTIndexPage` are provided for you. We recommend that you study the source code for these two classes carefully. Their implementation can be found in `src/btleaf.cpp` and `src/btindex.cpp`. You are free to add any methods to these classes that can help with your assignment (try to keep the methods private wherever possible). *Do not add any member variables to these classes since they have to map carefully onto the structure of a page.*

BTLeafPage

Each indexing entry that we insert or delete from a `BTreeFile` is a tuple (key, record id). These tuples are stored as records of type `LeafEntry` in `BTLeafPage`. To facilitate a scan through the leaf nodes, the leaf nodes are linked together as a doubly linked list.

These "pointers" in the link list are actually `PageID` of the previous page and next page. The two members corresponding to these pointers are called `nextPage` and `prevPage`. They can be set and retrieved with `SetNextPage()`, `SetPrevPage()`, `GetNextPage()`, and `GetPrevPage()`; see `include/heappage.h`.

Other functions provided for `BTLeafPage` are `Insert` and `Delete`, which (as the name suggest) insert and delete a `LeafEntry` to/from the page. `GetNext()`, `GetFirst()`, and `GetCurrent()` are also provided to scan through the records in a `BTLeafPage`. A typical use of these functions is shown below.

```
int key;
RecordID rid, outRid;
Status s = page->GetFirst(key, rid, outRid);
while (s != DONE)
{
    // do something with key and rid
    s = page->GetNext(key, rid, outRid);
}
```

BTIndexPage

The `BTIndexPage` contains a sequence $< pid_0, key_1, pid_1, \ldots, key_k, pid_k >$ (the semantics of this sequence are as described in the lectures and textbook). The `prevPage` member in `BTIndexPage` (also referred to as the left link or the leftmost child page) is used to store pid_0, while (key_i, pid_i) are stored as records of type `IndexEntry` in the `BTIndexPage`.

`Insert()`, `Delete()`, `GetFirst()`, and `GetNext()` are provided for this class.

Scanning a B+ Tree

We can also open a scan on a B+ tree and retrieve all records within a range of key values. The `BTreeFileScan` class is a data structure to keep track of the current cursor in the scan. It provides a `GetNext()` method for retrieving the next record in the `BTreeFile`. We can also delete the current record at the cursor while we are scanning, using the `DeleteCurrent()` method.

Assignment Requirements

In this assignment, you need to implement two classes: `BTreeFile` and `BTreeFileScan`, link them with the main test program, and make sure that all test programs run successfully. Note that a successful run does not mean that your program is correct. You should also ensure that your program will work for all possible test cases.

To simplify the assignment, you can assume that a BTreeFile only handles integer keys. Also, your implementation does not have to deal with duplicate key values.

The details of the functions that you have to implement are given below. We have also provided some sample code that illustrates the use of the classes we have provided. However, these samples are extremely simple and do not reflect the actual coding that we expect from you. For example, we do not show any error checking in these samples. You are expected to write robust programs by checking the return code and signal errors when necessary.

BTreeFile

- `BTreeFile::BTreeFile`

  The constructor for the `BTreeFile` takes in a file name, and checks if a file with that name already exists in the database. If the file exists, we "open" the file. Otherwise, we create a new file with that name.

  You can use `MINIBASE_DB->GetFileEntry()` to check if the file exists. A file entry is a pair `(filename,pid)`, where `pid` is the page id of the "first" page of the file. `GetFileEntry()` takes in a file name and returns the page id of the first page. In our case, the returned page id should corresponds to the page id of the root note. If no such file is found in the database, `GetFileEntry()` returns `FAIL`.

  To create a new B+ tree index, you should first make a new page, and initialize it. You should also use `MINIBASE_DB->AddFileEntry()` to add a new file entry into the database. The following example shows how to do this:

```
Page *page;
MINIBASE_BM->NewPage(pid, page);
MINIBASE_DB->AddFileEntry(filename, pid);
((SortedPage *)page)->Init(pid);
```

  After the above, you should also initialize the type of the page to the appropriate type (is it `LEAF_NODE` or `INDEX_NODE` ?).

```
((SortedPage *)page)->SetType(???);
```

  If `GetFileEntry()` returns `OK` and we get the page id of the root, we "open" the file by pinning the first page, which causes the root to be read from disk into memory. The following example shows how to read an existing file and pin the first page.

```
// filename contains the name of the BTreeFile to be opened
Page *page;
MINIBASE_DB->GetFileEntry(filename, pid);
MINIBASE_BM->PinPage(pid, page);
```

- `BTreeFile::~BTreeFile`

  The destructor of `BTreeFile` just "closes" the index. This includes unpinning any pages that are being pinned. Note that it does not delete the file.

- `BTreeFile::DestroyFile`

  This method deletes the entire index file from the database. You need to free all pages allocated for this index file. The file entry in the database also needs to be removed using `MINIBASE_DB->DeleteFileEntry(filename)`.

- `BTreeFile::Insert`

  This method inserts a pair `(key,rid)` into the B+ tree index. The actual pair `(key,rid)` is inserted into a leaf node. But this insertion may cause one or more `(key,pid)` pairs to be inserted into B+ tree index nodes. You should always check to see if the current node has enough space before you insert. If there is not enough space (node overflow), you have to split the current node by creating a new node

and copy some of the data over from the current node to the new node. Note that this could recursively go all the way up to the root, possibly resulting in a split of the root node of the B+ tree. Splitting will cause a new entry to be added in the parent node. In this assignment, you will implement node splitting for handling overflow on inserts. You do not need to implement sibling redistribution on inserts.

Splitting of the root node should be considered separately, since if we have a new root, we need to update the file entry to reflect the changes. Also, if you split a leaf node, be careful in maintaining the linked list of B+ tree leaf nodes.

Due to the complexity of this function, we recommend that you write separate functions for different cases. For example, it is a good idea to write a function to insert into a leaf node, and a function to insert into an index node. The following shows some simplified code fragment that may be helpful.

Checking if there is enough space to insert a record into a leaf node:

```
if (page->AvailableSpace() ...
```

Inserting a pair `(key,rid)` into a leaf node with page id `pid` can be done with the following code:

```
BTLeafPage *page;
RecordID outRid;
MINIBASE_BM->PinPage(pid, (Page *)page);
page->Insert(key, rid, outRid);
MINIBASE_BM->UnpinPage(pid, DIRTY);
```

- `BTreeFile::Delete`

  This method deletes an entry `(key,rid)` from a leaf node. Deletion from a leaf node may cause one or more entries in the index node to be deleted. You should always check if a node underflows (less than 50% full) after deletion. If so, you should perform sibling redistribution, and if that does not work, merge sibling nodes (read and implement the algorithm in the textbook).

  For implementing redistribution in this assignment, it is sufficient for you to pick one of the two siblings and try to redistribute. For example, you can always pick the left sibling to perform redistribution; if that fails, merge sibling nodes.

  You should consider different scenarios separately (maybe write separate functions for them). You should consider deletion from a leaf node and index node separately. Deletion from the root should also be consider separately (what happens if the root becomes empty after some deletion, but there are still some child nodes?)

  The following code fragment may be helpful.

  Checking if a node is half full:

```
if (page->AvailableSpace() > HEAPPAGE_DATA_SIZE / 2)
{
    // check if merge can occur
    // merge
}
```

- `BTreeFile::OpenScan`

  This method should create a new `BTreeFileScan` object and initialize it based on the search range specified. It is useful to find out which leaf node the first record to scan is in.

- `BTreeFile::DumpStatistics`

  This method should print out the following statistics of your B+ tree.

  1. Total number of nodes in the tree.
  2. Total number of data entries in the tree.
  3. Total number of index entries in the tree.
  4. Average, minimum, and maximum fill factor (used space/total space) of leaf nodes and index nodes.
  5. Height of tree.

  These statistics should serve you in making sure that your code executes correctly.

- `BTreeFile::Print`, `BTreeFile::PrintTree`, `BTreeFile::PrintNode`

  These are helper functions that should help you debug, by showing the tree contents. The implementation has been provided. You can modify it to suit your own needs.

BTreeFileScan

First, note that `BTreeFileScan` only has a default constructor, which you do not need to implement. Also, a `BTreeFileScan` object will only be created inside `BTreeFile::OpenScan`.

- `BTreeFileScan::~BTreeFileScan`

  The destructor of `BTreeFileScan` should clean up the necessary things (such as unpinning any pinned pages).

- `BTreeFileScan::GetNext`

  `GetNext` returns the next record in the B+ tree in increasing order of key. Basically, `GetNext` traverses through the link list of leaf nodes, and returns all the records in them that are within the scan range, one at a time.

  For example, if the B+ tree contains records with keys $1, 2, 4, 5, 6, 7, 8$, the following code should print out "2 4 5".

```
int low = 2;
int high = 5;
RecordID rid;
int key;
scan = (BTreeFileScan *)tree->OpenScan(&low, &high);
scan->GetNext(rid, key); cout << key << "␣";
scan->GetNext(rid, key); cout << key << "␣";
scan->GetNext(rid, key); cout << key << "␣";
```

- `BTreeFileScan::DeleteCurrent`

  `DeleteCurrent` should delete the current record, i.e, the record returned by previous call to `GetNext()`.

  For example, if the B+ tree contains records with keys $1, 2, 4, 5, 6, 7, 8$, the following code should print out "2 4", and the remaining B+ tree should contain records with keys $1, 5, 6, 7, 8$.

```
int low = 2;
int high = 5;
RecordID rid;
int key;
scan = (BTreeFileScan *)tree->OpenScan(&low, &high);
scan->GetNext(rid, key);
cout << key << "␣";          // should output 2
scan->DeleteCurrent();       // record with key 2 is deleted
scan->GetNext(rid, key);
cout << key << "␣";          // should print 4
scan->DeleteCurrent();       // record with key 4 is deleted
scan->DeleteCurrent();       // error since record with key 4
                                        // is already deleted
```

Documentation

You should also submit a document describing the code that you have written. This document will be especially helpful if you submitted code does not pass all our test cases. This document should include assumptions that you made, descriptions of any new classes that you have added, and any other special feature we should take note of. As a guideline, the document should be 2-3 pages long (with normal fonts and spacing), or more, if you feel necessary. If you did not complete the assignment, explain the part that you completed so that we can give partial credits (e.g., deletion works, except that it will cause the program to fail if an index entry is deleted from the root node). Show sample test runs that you have used to ensure your code is correct, and explain why they are meaningful.

Getting Started

The file `BTree.tgz` contains the skeleton code and libraries necessary for this assignment. The libraries are precompiled for the DICE environment, thus we expect that you develop and test your solution on a DICE machine. The grading of this assignment will also be done on a DICE machine.

Assuming your working directory is `~/`, copy the file `BTree.tgz` into your working directory and unzip it using the command `tar -xvzf BTree.tgz`. This will result in a folder `~/BTree/` containing the source code and makefile of the project.

`src/btfile.cpp`, `include/btfile.h`	code skeleton for `BTreeFile` class
`src/btleaf.cpp`, `include/btleaf.h`	code for `BTLeafPage` class
`src/btindex.cpp`, `include/btindex.h`	code for `BTIndexPage` class
`src/btfilescan.cpp`, `include/btfilescan.h`	code skeleton for `BTreeFileScan` class
`src/btreetest.cpp`, `include/btreetest.h`	code for test-script interface to the B+-tree
`src/sortedpage.cpp`, `include/sortedpage.h`	code for `SortedPage` class
`include/bt.h`	general declaration of types
`src/main.cpp`	contains `main()`, runs tests

You should write most of your code in `BTreeFile` (.h and .cpp) and `BTreeFileScan` (.h and .cpp). You can add any member variables and methods to these two classes. Again, you can also add new methods to the `BTLeafPage` (.h and .cpp) and `BTIndexPage` (.h and .cpp) classes (keep these methods private wherever possible). You *should not* modify any of the other existing methods in `BTLeafPage` and `BTIndexPage`. You also *should not* add any member variables to these classes, since they have to map carefully onto the structure of a page.

To compile the source code, run `make`. This should finish without errors. To re-compile the source code after having modified parts of the code, it suffices to run `make` (or run: `make clean; make`). The resulting executable file is `bin/btree`.

Test Interface

You can run the program either interactively, or using a script. For a list of accepted commands, execute "`bin/btree ?`".

- You can execute the program standalone, in which case the program will wait for commands on stdin.
- You can execute the program as `bin/btree <file>`, in which case the program will read commands from `<file>`. A file named `test.txt` has been provided as a sample, but it is far from a complete test set.

Below are the commands accepted. Note that when `<low>` and/or `<high>` equals -1, they mean min and max, respectively.

`insert <low> <high>`	Insert all the records from [low, high] sequentially
`scan <low> <high>`	Scan for records in [low, high]
`delete <low> <high>`	Delete records in [low, high]
`deletescan <low> <high>`	Scan and deleteCurrent records [low, high]
`print`	Print B+ tree
`stats`	Show statistics
`quit`	Termination

What to Turn In

You are required to turn in exactly the same set of source code files as the set of files given to you at the beginning of the project: no more files, and no fewer files. The document describing your effort and the source code files should be zipped up into a file named `BTree.tgz` (or `.zip`) using the command `tar -cvzf BTree.tgz BTree <document>`.

The source code files should be organized in the same directory structure as well. Again, you should write most of your code in `BTreeFile` (.h and .cpp) and `BTreeFileScan` (.h and .cpp). You can add any member variables and methods to these two classes.

Upload your solution (i.e., tgz/zip file) using the LEARN submission system: Assessment (the left panel) → Assignment Solution → ADBS Coursework: B+ Tree.

Grading Policy

We will grade your program based on the following criteria:

- Correctness (80%): You will get full marks if your implementation is correct. Partial credit will be given to a partially correct submission.
- Documentation (20%): Documentation should be short, clear, concise, and informative. It should contain enough material so that one can understand the code better after having read it.

Important Advice

- Do this assignment in increasingly more difficult steps. For example, you might want to implement `BTreeFile::Insert` for tree with only one node (a root) and assume no overflow. Then implement `BTreeFile::Insert` that handles overflows in leaf nodes, and then implement `BTreeFile::Insert` to handle overflows in index nodes.
- Be careful and remember to unpin any pages that you pin. Otherwise your buffers may become full and you will not be able to pin any more pages. Or you might get errors when trying to unpin pages that are not pinned.

Make sure you start early! Good luck!

FAQ

Q: (`BTreeFile` constructor) For the `BTreeFile` constructor, when we are creating the root node, should it be a leaf node or an index node?

A: If there is only the root node, it should be of type leaf node.

Q: (`BTreeFile` constructor) How do I "open"the `BTreeFile`? Should I use `fopen()`?

A: Use `MINIBASE_DB->GetFileEntry()` to get the id of the root page, and then pin the page into the buffer. You should not use `fopen()`. The lower layer takes care of opening and closing files. Look at the methods in `include/db.h`.

Q: (`BTreeFile` destructor) In the destructor for `BTreeFile`, how do we "close" the index file?

A: Just unpin all pages. Be careful to update the entry for `MINIBASE_DB` when the root changes (does not have to be in the destructor, but you have to take care of it somewhere).

Q: (Search) How can I find the child nodes of a BTIndexNode?

A: The pointers to the child nodes are stored in records that you design. Each record consists of the pair `(key,pid)`. The leftmost pointer is stored in `prevPage`.

Q: (Insertion) For insertions, do I have to implement both splits and redistribution?

A: No. You have to implement splits, but not redistribution.

Q: (Deletion) For deletions, do I have to implement both merging and redistribution?

A: Yes, you have to implement both strategies.

Q: (Deletion) What is the minimum occupancy? Is it half the maximum capacity?

A: You can use the function `IsAtLeastHalfFull()` that is defined in `BTLeafPage` and `BTIndexPage`. You can modify this function if necessary.

Q: (`BTreeFileScan`) Assume that the B+ tree contains the key values $4, 5, 7, 9, 13$. What should `GetNext()` return if I call `OpenScan(6,10)`? What if I call `OpenScan(10,12)`?

A: If you call `OpenScan(6,10)`, `GetNext()` should first return 7, then return 9, then return `DONE`. If you call `OpenScan(10,12)`, then the first call to `GetNext` should return `DONE`. You have to return a `DONE` object, even if there are no keys in the specified range.

Q: (`BTreeFileScan`) What should we initialize in `OpenScan`?

A: You should initialize some member variables to keep track of the current state (including the high key and the low key), so that you can actually perform the scan using `GetNext()`.

Q: (`BTreeFileScan`) In `BTreeFileScan::DeleteCurrent()`, do we have to do a deletion with merging and redistribution, as in `BTreeFile::Delete`?

A: Yes.

Q: (`BTreeFileScan`) What should I return when `BTreeFileScan::DeleteCurrent()` is called on the last record in the specified range?

A: You should return `OK`. If `DeleteCurrent()` is called a second time at same position, without a call to `GetNext()` in between, return `FAIL`.

Q: Are we allowed to modify `SortedPage` and `HeapPage`?

A: No.